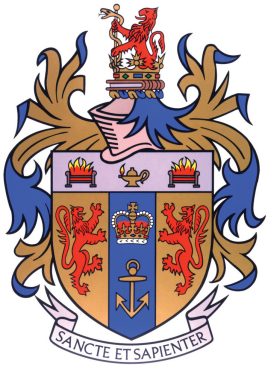
Editing composed languages

Diekmann, Lukas

Awarding institution:
King's College London

Editing composed languages



Lukas Diekmann

Thesis submitted in partial fulfilment for the degree of
Doctor of Philosophy

Faculty of Natural & Mathematical Sciences
Department of Informatics
King's College London

June 2019

Abstract

The development of domain-specific languages and the migration of legacy software have long been problems for which language composition offers an enticing solution. Unfortunately, approaches thus far have failed to meet expectations, largely due to the difficulty of writing composed programs. Language composition editors have traditionally fallen into two extremes: traditional parsing, which is inflexible or ambiguous; or syntax-directed editing, which programmers dislike. This thesis extends an incremental parser to create an approach that bridges the two extremes: an editor that 'feels' like a normal text editor, but always operates on a valid tree as in syntax-directed editing, which allows users to compose arbitrary syntaxes. I first take an existing incremental parsing algorithm and fix several errors in it. I then extend it with incremental abstract syntax trees and support for whitespace-sensitive languages, which increases the range of languages the algorithm can support. I then introduce the notion of language boxes, which allow an incremental parser to be used for language composition and implement them in a prototype editor. Finally, I show how language boxes can, in many useful cases, be automatically inserted and removed without the need for user intervention.

Acknowledgements

I would first like to thank my supervisor Laurence Tratt, who, despite encouraging me to write this thesis, has become a good friend throughout the last few years. Without his help this thesis would not have been possible and I am extremely grateful for his support and advice in both academic as well as non-work related matters.

I would also like to thank my friends and colleagues at King's for making life and work there more enjoyable: Josef Bajada, Edd Barrett, Carl-Friedrich Bolz, Martin Chapman, Elliot Fairweather, Christopher Hampson, Jake Hughes, Sarah Mount and Gareth Muirhead. Special thanks goes to Naveneetha Vasudevan for reviewing parts of this work. Many thanks also goes to Malte Seifert for being a lifelong friend who I can always count on.

To my family, who have supported me throughout my whole life and have continued to support me in every way they could, even after my move to the UK – Mama, Papa, Jan, Eva, thank you. I love you all. To my wife Clara for her unending love and support, who cluelessly proofread parts of this work – I love you. To my son Emil, without whom this thesis probably would have been completed a few months earlier – despite only having known you a bit over two months, I already love you dearly, even when your insatiable need for food and attention keeps your parents awake at night.

Contents

Chapter 1

Introduction

Language composition has long been an enticing solution to problems such as DSL development and migrating legacy software. However, current approaches have failed to make this promise a reality because they struggle with the problem of writing composed programs. In this thesis I develop a novel approach to editing composed programs, based on incremental parsing, which bypasses many of these problems.

1.1 Motivation

Due to the popularity of general purpose languages, they are often a user's first choice over domain specific languages (DSLs), even if the DSL would be better suited for the task. Unfortunately, once the choice for a particular language is made, it is difficult to branch out to others as there is a wall between different languages that makes it difficult to use them in conjunction. Most languages have thus been extended with interfaces to popular DSLs, such as SQL, which almost all major general purpose languages support. However, such extensions are often crude (e.g. embedding via strings) and the languages remain syntactically isolated from each other, limiting interactivity between them and making them less safe (e.g. injection attacks).

Another growing problem is that of legacy software. Many important software systems that we rely on today were written in the late 70s and 80s using languages that are now considered outdated. Attempts to replace such legacy systems with equivalents written in modern languages are often unsuccessful [30, 31, 34], as it requires swapping out the old system in one go. A better approach would be to gradually replace small parts of the system and test each change rigorously before moving on to the next, until the entire system has been successfully rewritten, thus making it easier to avoid nasty

surprises when the new system goes live. Unfortunately, this is not possible with current technologies.

1.2 The problems with language composition

A partial solution to these problems is offered by language composition, or more specifically, one form of it which extends languages by embedding one language into another. The idea of language composition has been around since the late 60s [16] but has not managed to gain widespread popularity apart from settings such as HTML/JavaScript or foreign function interfaces (FFI). A major reason for this is that syntactically composing programming languages is non-trivial. This makes editing composed programs difficult.

Programs are typically edited and executed in two different ways, either via a parsing-based approach using a normal text editor, or through the use of syntax-directed editors (SDE). Unfortunately, using traditional parsers for language composition can be difficult to deal with: composing two LR grammars, in general, leads to a non-LR grammar [63]; generalised parsing approaches often lead to ambiguities, which are difficult to detect [15]. To solve this problem, parsing-based approaches typically use separators between languages to disambiguate their syntax. Separators, however, are inflexible as they must not appear in any of the composed languages.

Syntax-directed editors follow a different philosophy for editing programs. Instead of allowing the user to type programs freely, syntax-directed editors create parse trees from predefined templates that the user fills in with more specific information. For example, when wanting to write a for-loop in a syntax-directed editor, the user selects the loop from a list of statements[1] which are valid at the current program location. The editor then inserts the whole construct into the parse tree, including condition and block delimiters (if needed), leaving only the bounds of the condition and the name of the index variable for the user to fill in. This ensures that there is always a valid parse tree and makes it impossible to create syntax errors. A useful side-effect of this is that it completely avoids ambiguity issues. This makes syntax-directed editors excellent tools for language composition. However, the restrictive nature of syntax-directed editors – text must be edited relative to an abstract syntax tree (AST) which prohibits certain operations such as selecting text across AST elements – can make editing such programs a frustrating experience and may alienate some users, especially those experienced with more traditional text editors [42].

[1] Modern syntax-directed editors have managed to improve the user experience by partially imitating traditional text editors. Statements need not to be chosen from a list but can be typed in and are automatically completed once the editor recognises them. For example, typing in `for` would automatically create the entire statement.

1.3 A new approach

The aim of this thesis is to find a practical solution for language composition that doesn't suffer from the problems that affect traditional grammar composition and syntax-directed editors. While there are existing parsing-based approaches to language composition, they either have problems with ambiguities (see Section 2.3), or limit the expressiveness of the composed languages, for example by introducing separators which must not appear in any of the languages (see Section 7.4). This section outlines an alternative parsing-based approach which allows the composition of languages without introducing ambiguities or restricting languages, while providing information about the program back to the user in as close to real-time as possible.

1.3.1 Requirements

Since the composition of two grammars typically results in ambiguities, many approaches solve this problem by introducing special markers which separate the languages (e.g. [91, 12]). Others (e.g. Copper [82, 70]) solve ambiguities between languages by prioritising one language over the other. Neither solution is ideal as they restrict one or more of the composed languages, i.e. special markers must not appear in any of the composed languages, and the prioritisation of one language requires limiting the expressiveness of the other. Such solutions are also language specific and need to be defined for each composition separately. An alternative solution to these problems is to use non-text-based separators between languages, which by definition cannot exist in any of the languages. This can be achieved by using an editing approach that is not dependent on the textual representation of the program, for example, by editing a parse tree directly, similar to syntax-directed editing.

In a traditional, batch-oriented parser programs are always parsed from top to bottom, generating a new parse tree every time the program is changed. With increasing program sizes this can quickly lead to performance problems. Many editors that provide features such as syntax highlighting, name binding analysis, or code completion thus employ heuristics, which don't require an always up-to-date parse tree. The downside is that such heuristics are often inaccurate, which can, for example, lead to faulty syntax highlighting or unnoticed name binding errors. Working on an always up-to-date parse tree instead achieves much better results. Using a parse tree also makes the implementation of such features much easier than developing and carefully refining heuristics. Additionally, heuristics are typically hand-written for a particular language, whereas the analysis of a parse tree can be generalised to work with multiple languages.

1.3.2 Solution

A solution to both of these problems is incremental parsing. Firstly, an incremental parser works on parse trees instead of plain text, which enables the use of non-textual special markers to separate languages and avoid ambiguity. Secondly, an incremental parser can find and re-parse only those parts of a program that have changed, while reusing results from previously generated parse trees. This allows such parsers to be used "online", i.e. continuously re-parsing user changes, even in large programs, as the user types, resulting in an always up-to-date parse tree which allows instantaneous and accurate feedback to the user. Incremental parsing algorithms exist for a variety of grammar types, e.g. LL or PEG (see Section 2.10). However, the largest class of unambiguous grammars that we can define is LR. Also, due to the popularity of parser generators such as Yacc, LR grammars for many popular programming languages, such as Java, PHP, SQLite, and Go, already exist. This makes LR a good choice for language composition.

In this thesis I propose a new solution for language composition, which avoids ambiguity issues, while keeping program editing as flexible as it is in normal text editors. The solutions presented in this thesis are implemented within a new programming editor *Eco*[2]. The approach I propose uses an incremental parsing algorithm created by Tim Wagner [88] as its core, correcting and extending it in various ways. Though originally these algorithms were intended for parsing individual languages, this thesis shows how they can be extended to create a new approach for editing composed programs. First I correct several problems in Wagner's algorithm as well as providing explanations for some vague or missing parts. I then show how the algorithm can be extended with concepts such as incremental abstract syntax trees and support for whitespace-sensitive languages; the former paves the way for IDE features such as syntax-highlighting and name binding; the latter increases the range of languages the algorithm can support. I then introduce the novel concept of language boxes[3], which allow the incremental parsing algorithm to be used for language composition. Finally, I show that in many useful cases, language boxes can be automatically inserted and removed by the editor, without user intervention.

1.4 Overview

Incremental parsers are based on traditional parsing algorithms. However, instead of parsing a whole file each time the program is executed, incremental parsers generate and update a parse tree from the user input as the user types. When editing a program, this

[2]The prototype editor *Eco* can be downloaded from `https://github.com/softdevteam/eco`

[3]The 'language boxes' in this thesis should not be confused with the modular language definition concept of the same name from [66].

parse tree is then modified directly, e.g. changing the name of a variable directly updates the value of the token in the parse tree that represents that variable. After each edit the incremental parser updates and re-parses the relevant parts of the parse tree, attempting to reuse as many unchanged parts of the tree as possible. This is not only faster than parsing the entire program from scratch, but also has the added benefit that meta-data on parse tree nodes is kept intact when the program is re-parsed.

Editing programs this way allows us to embed languages into one another without separators, by using language boxes. These boxes can be manually inserted by the user to write code in another language. When creating a new language box, it is inserted directly into the parse tree. Each language box has its own parse tree which is maintained by its own incremental parser. This way languages stay completely separated from each other which bypasses all ambiguity problems. For example, when editing a program in language *X*, the user can at any point insert a language box for language *Y*. Typing inside of that box edits in language *Y*, while typing outside of the box edits in language *X*. Programs may contain any number of languages boxes, and language boxes can be nested arbitrarily deep. While the main focus of this thesis is the editing of textual languages, language boxes are not limited to it: I thus also briefly explore the use and composition of non-textual languages.

This thesis also shows how an incremental parser can be extended to produce incremental ASTs. ASTs are used to compile programs into executable code but are also needed to semantically analyse programs, giving users additional information about the correctness of their code (name binding), formatting it to make it more aesthetic (syntax-highlighting), or guiding them to make writing code easier (code completion). Generating ASTs and analysing them is time consuming, and is thus run as a subprocess in many editors. This can often lead to noticable delays when using IDE features such as code completion, for example. Generating ASTs incrementally improves the performance of such features, and opens the door for incremental approaches, which can reduce these delays even further.

Unfortunately, some languages don't naturally work with an incremental parser. An example for this is Python [81], which uses indentation to mark blocks. This typically requires an additional phase to generate indentation tokens into the sequence of tokens generated by a lexer. Making this phase incremental is thus also explored in this thesis.

Even though language boxes fit naturally within an incremental parser, they still come with some additional challenges. One is the interaction of language boxes with other tokens such as comments and strings. This thesis thus explores how an incremental lexer can be extended to support the embedding of language boxes within other tokens without having to flatten the language box into normal text. Another is that language

boxes need to be inserted manually by the user, which in some cases can seem tedious and unnecessary. This thesis thus also explores how language boxes can be inserted automatically and without the need for user interaction whenever possible.

1.5 Contributions

The main contributions of this thesis are as follows:

1. Identifying and fixing flaws in Wagner's incremental parsing algorithm.
2. Extending the fixed incremental parsing algorithm with incremental abstract syntax trees and support for whitespace-sensitive languages such as Python.
3. Introducing the novel concept of language boxes, showing that they allow fine-grained syntactic language composition.
4. An extension to language boxes that allows them to be automatically created in many realistic circumstances.

1.6 Synopsis

The structure of this thesis is as follows: Chapter 2 gives a brief overview of popular parsing techniques and the problems that arise when they are used in combination with language composition. It also provides background information about the incremental parsing techniques developed by Tim Wagner [88] and extends that information with details that Wagner's thesis only touches upon or that are ambiguously explained. Chapter 3 discusses Wagner's error recovery algorithms, explains issues with their implementation, and proposes fixes and some optimisations. Chapter 4 shows extensions to Wagner's incremental parsing techniques, adding support for incremental abstract syntax trees and whitespace-sensitive languages such as Python. Chapter 5 shows the editor *Eco* which was produced alongside this thesis and explains some finer details of its implementation. The main focus of that chapter is on language boxes, one of the major contributions of this thesis. It explains how language boxes can be integrated into an editor and shows how they can be useful for language composition of textual as well as non-textual languages. The chapter also discusses the testing framework used in *Eco* and shows how employing fuzz testing helped ridding *Eco* of bugs that would otherwise have been difficult to find. Chapter 6 discusses the problem of embedding language boxes into strings and comments and proposes a solution which has been implemented in *Eco*.

Chapter 7 shows an (optional) extension for language boxes: automatic language box detection. This extension aims to streamline the use of language boxes by inserting them automatically, wherever possible, without user interaction and without imposing on the user's ability to type freely. Finally, Chapter 8 discusses possible further extensions to *Eco* and incremental language technology in general that were out of the scope of this thesis.

1.7 Publications

The majority of Chapters 4 and 5, and parts of Chapter 2, have appeared in an identifiable form in the following publications of which I was the main author:

- Lukas Diekmann and Laurence Tratt. *Parsing Composed Grammars with Language Boxes.* Workshop on Scalable Language Specification, June 2013
- Lukas Diekmann and Laurence Tratt. *Eco: A language composition editor.* In SLE, pages 82–101, September 2014.

Some parts of Chapter 5 have appeared in the following publication, where I was the main author of parts relevant to *Eco* and secondary author otherwise:

- Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. *Fine-grain language composition.* In ECOOP, July 2016.

Chapter 2

Background

This chapter serves as a refresher for concepts the reader should be familiar with before tackling the remaining chapters and gives an overview of the techniques which this thesis is largely based on. Sections 2.1, 2.2, and 2.3 describe typical parsing techniques and explain how they compare in the context of language composition. Sections 2.4, 2.5, and 2.6 give an overview of the incremental parsing techniques developed by Tim Wagner [88], some of which include my own implementations wherever Wagner didn't provide his own (all algorithms in this thesis are marked with subscripts, i.e. W for Wagner's original algorithms, Wb for implementations based on Wagner's descriptions, WD for versions of Wagner's algorithms that have been fixed or optimised by me, and D for my own original algorithms).

2.1 Context-free grammars

Most non-natural languages (i.e. programming or markup languages) have an underlying context-free grammar that describes their syntax and can be used to decide if a program is valid in that language or not. In the remainder of the thesis we will call context-free grammars *CFGs* or just *grammars*.

Unless stated otherwise, all grammars in this thesis use the same notation as Yacc [38] (a popular and widely used parser generator). A grammar consists of one or more *rules*. The left-hand-side of a rule describes its name and can be used to reference it within other rules. The right-hand-side of a rule consists of one or more *productions* (often called 'alternatives') which are separated by the character '`|`'. Each production consists of a sequence of symbols; a symbol is either a *nonterminal* referencing another rule from the grammar (one of which is chosen as the *start rule*), or a *terminal*, i.e. a token type such

```
// Grammar rules
%start E
%%
E: T
 | E "add" T ;
T: P
 | T "mul" P ;
P: "int" ;
```

```
// Lexing rules
add:"\+"
mul:"\*"
int:"[0-9]+"
```

Listing 2.1: An example of a simple grammar for a basic calculator language. The grammar rules are shown on the left; lexing rules on the right. The separation of multiplication and addition into two rules `T` and E is to allow multiplication to take priority over addition. In Yacc, the start rule can be declared explicitly. Alternatively, Yacc chooses the first rule in the grammar as the start rule. The explicit use of '`%start`' is therefore omitted for the remainder of this thesis.

as `INT`. Lexing rules are of the form `ID: REGEX` and are considered in the order in which they are defined to avoid longest-match ambiguities. Listing 2.1 gives an example of a simple grammar for a calculator language.

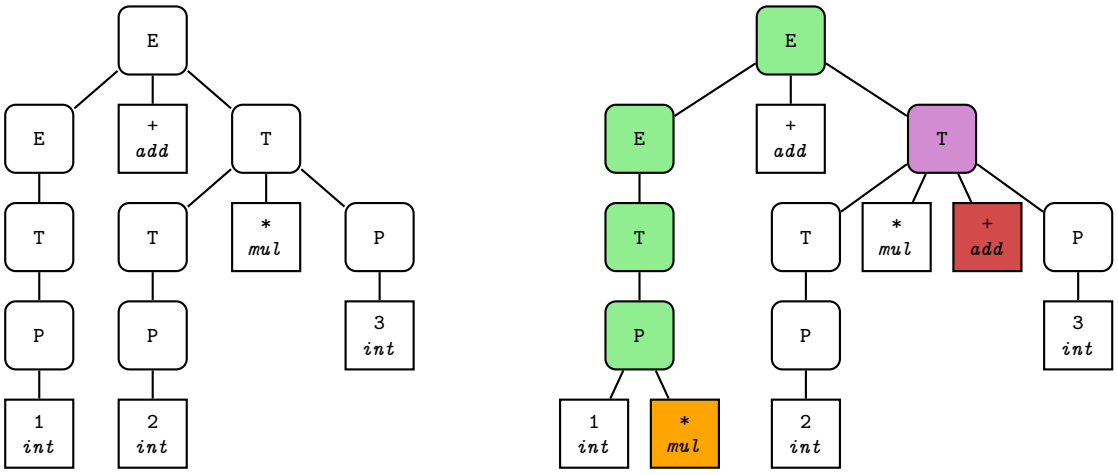
2.2 Parse trees

In order to parse a program, it is first split into tokens by a lexer according to some lexing rules. For example, a lexer that knows about arithmetic would likely split the program '`1+2`' into three tokens (`1`, `INT`), and (`+`, `add`), (`2`, `INT`), where each token is of the form (*value*, *type*). A parser then takes the stream of tokens as input and decides if the program is valid with respect to the given language. Parsers can be either written by hand or generated from a grammar. The result of a parser is typically a tree representation of the program, called a *parse tree*. Figure 2.1a shows a parse tree, constructed using the calculator grammar from Listing 2.1.

A parse tree consists of tokens, representing input generated by the user, and nonterminals, representing grammar rules, which are generated by the parser. Nonterminal nodes have a type but no value and can have other nonterminals and tokens as children or be completely empty. Tokens have a value and a type but cannot have children at all. Unless stated otherwise, all parse trees shown within this thesis will use the calculator grammar from Listing 2.1. They also follow a consistent colouring scheme, shown in Figure 2.1b, which aids in the understanding of some of the algorithms presented later on in this thesis.

2.3 Parsing and syntax-directed editing

Traditionally, programs are created by typing text into an editor and saving it to a file, which is then processed by a parser. While there are many possible approaches to parsing



(a) Parse tree **(b)** Colouring scheme

Figure 2.1: (a) A parse tree generated from the input '`1+2*3`' using the calculator grammar from Listing 2.1. Tokens are represented as rectangles, while nonterminal nodes are represented as rounded rectangles. **(b)** The parse tree colouring scheme used throughout this thesis. Subtrees containing changes are coloured in green. New tokens, inserted by the user, and new nonterminals generated by the parser (not shown) are coloured in orange. Nodes and subtrees containing errors are red. Isolated nodes, which are introduced in Chapter 3, are purple.

text, three approaches can be used as exemplars of the major categories: LR parsing, generalised parsing, and PEG parsing. A different technique is syntax-directed editing, where the user is guided by the editor, which removes the need for parsing. This section briefly explains the different techniques and to which extent they can be used for language composition.

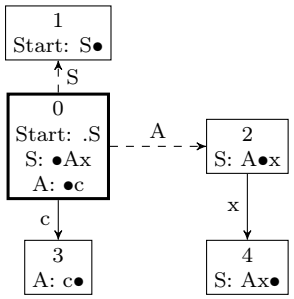
2.3.1 LR parsers

Due to Yacc's predominance, LR-compatible grammars are commonly used to represent programming languages. Indeed, many programming language grammars are deliberately designed to fit within LR parsing restrictions.

LR parsers [44, 1] are typically created through the use of parser generators. The generator reads a grammar and creates a state graph from it. Each state in the graph references one or more rules. The edges between states are labelled with the input the language can understand. From this graph the generator then constructs a parse table. A parse table is, in essence, a more memory efficient representation of the state graph that simplifies the processing of inputs. Figure 2.2 shows an LR grammar alongside its state graph and parse table.

Once the parse table has been generated the state graph can be discarded. The parse table is used by the parser to decide if some input is valid for a given grammar and to construct a parse tree from it. When parsing input the parser uses a parsing stack to

```
%start S
%%
S: A "x"; // (I)
A: "c";   // (II)
```



	Actions			Gotos	
State	`c`	`x`	`$`	S	A
0	S(3)			1	2
1			Accept		
2		S(4)			
3		R(II)			
4			R(I)	6	

Figure 2.2: When creating a parser from a grammar (left), the parser generator first constructs a state graph (middle). The graph starts at state 0. Solid edges between states describe the shifting of terminals, dashed lines describe Goto transitions from reductions. Each item within a state $[N : \alpha \bullet \beta]$ references one of rule N's productions; α and β each represent zero or more symbols; with the *dot* ($\bullet$) representing how much of the production must have been matched (α) if parsing has reached that state, and how much remains (β). The state graph is then transformed into a parse table (right): S(x) means 'shift to state x'; R(y) means 'reduce grammar production y'. Empty cells in the table represent errors. The `$` terminal is special and automatically added by the parser; it describes the end of the file.

store the parse tree and current state. The stack is initialised with the end-of-file symbol and state 0, e.g. (`$`, 0). The parser then reads a token from the input and, using its type, looks up its action for the current state from the parse table. The currently processed token is also called *lookahead*. The parse table returns *Shift(x)*, *Reduce(y)*, *Error*, or *Accept*. Shift(x) pushes the lookahead onto the stack, moves to state x and sets the next token from the input as the new lookahead. Reduce(y), where y is of the form $N : \alpha$, pops $|\alpha|$ elements from the stack, uses the current state on the stack to look up the goto state for N and moves to the resulting state. It then creates a new nonterminal N with the popped elements as children, and pushes it onto the stack. When the parse table returns *Accept* the input has been successfully parsed.

For the grammar in Figure 2.2 the parser would parse the input "`cx`" as follows: We start in state 0 and the first token is '`c`'. The parse table returns S(3), which means we shift the lookahead '`c`' onto the stack and move to state 3. The next lookahead is '`x`' for which the parse table returns R(II). According to the rule, we pop one element from the stack which sets the parser back to state 0. We then look up `A`'s Goto state, which is 2, and move there. Then we create a new nonterminal `A` with child '`c`' and push it onto the stack. The lookahead is still '`x`', however, the parser is now in state 2, for which the parse table returns S(4), so we shift the lookahead onto the stack and move to state 4. Since the input is now empty, the next lookahead is '`$`', for which the parse table returns R(I). We pop the elements '`x`' and '`A`' from the stack, move to state 1 and push a new nonterminal '`S`' with children '`x`' and '`A`' onto the stack. Finally, the parse table returns *Accept* and the input has been successfully parsed.

```
// Grammar 1
%start S1
%%
S1: C "x" ;
C: "w" ;

// Grammar 2
%start S2
%%
S2: B ;
B: "w" "x" ;
```

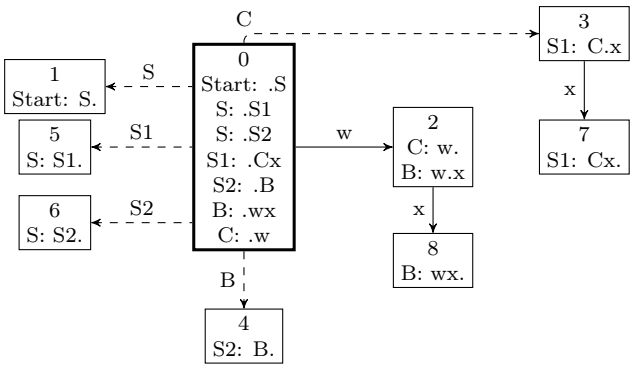


Figure 2.3: State graph after composing two grammars through union (`S: S1 | S2`). Generating a parse table from this graph results in a Shift/Reduce conflict for state 2 and symbol '`x`'. The symbol can be processed by either directly shifting '`x`' resulting in state 8, or by first reducing `C: w`, resulting in state 3, and then shifting '`x`' to reach state 7.

Even though LR(1) is more powerful than LALR (i.e. LALR only accepts a subset of the languages that LR(1) can accept), most parser generators, including Yacc, use the latter. Since most programming languages can be expressed in LALR, and LR(1) tables are exponentially bigger than LALR tables, this was an obvious choice for the performance and memory lacking computers of the 1970s [19, 3]. Nowadays, computers are fast enough to make LR(1) parser generation feasible, especially when paired with techniques that significantly reduce the size of LR(1) state graphs [62].

Unfortunately, composing two LR grammars does not, in general, result in a valid LR grammar [63]. A simple example for this is the composition of the two languages `A: 'a'` and `B: 'a'`, which is clearly ambiguous since we cannot know whether we should parse the input '`a`' in language *A* or language *B*. A slightly less obvious example is shown in Figure 2.3. Composing the two LR(1) grammars through union, e.g. `S: S1 | S2`, results in an ambiguous grammar which can be seen in state 2. When parsing the input '`wx`', the parser cannot decide whether '`x`' should be shifted immediately, or if a reduction should be applied first, since both are valid actions. When generating a parse table from this state graph, this would thus result in a Shift/Reduce conflict, as LR parse tables only allow one action per cell.

One partial solution to this is embodied in Copper which, by making the lexer lazy and context-sensitive, is able to allow many compositions which would not normally seem possible in an LALR parser [82, 70]. However, this requires nested languages to be delineated by special markers, which is visually obtrusive and prevents many reasonable compositions.

2.3.2 Generalised parsing

Generalised parsing approaches such as [23, 71, 85] can accept any CFG including inherently ambiguous grammars, which makes them a good choice for language composition. When parsing ambiguous input, the parser follows each option and generates multiple parses of the same program. A generalised parser constructed from the composed grammar in Figure 2.3 would be able to parse the program '`wx`' by following both options and producing two distinct parse trees (or a single parse forest where subtrees are shared between multiple parse trees) as a result. Though parsing ambiguous grammars will always be less efficient than parsing unambiguous ones, GLR runtime has been greatly improved [39] and measuring the effects of ambiguity on practical languages has shown that its impact on speed in such cases is lower than feared [89, 54].

Unfortunately, programming language tools (e.g. editors) and ambiguity don't work well with one another since the latter can hardly ask the user which parse (of possibly many) they intended. To make matters worse, two unambiguous grammars, when composed, may become ambiguous. We know that ambiguity is undecidable [15], i.e. the only way to determine CFG ambiguity is to test every possible input, which is impractical since most CFGs describe infinite languages. Although heuristics for detecting ambiguity exist, all existing approaches fail to detect at least some ambiguous grammars [83].

Traditional parsers, including many generalised parsers, run lexing and parsing as two separate phases. Scannerless parsers, a variant of generalised parsers, on the other hand intertwine the two [86]. This allows them to automatically solve lexical ambiguities by context. For example, the grammar `S: 'x' 'a' 'b' | 'y' 'ab'` is problematic for a parser with a separate lexing phase. While we can create a conflict-free LR parser from it, the lexing phase doesn't have enough information to determine if the input '`ab`' should be tokenised as two tokens '`a`' and '`b`', or a single token '`ab`' (a lexer will usually pick the longest match, which in this case would be '`ab`'). A scannerless parser doesn't have this problem and will automatically choose the tokenisation that is valid in the current context. If it previously parsed '`x`', then the only valid character following this is '`a`'. If however the first character was '`y`', then only '`ab`' is valid at this point. In essence, a scannerless parser looks at the grammar first and then tries to complete it by generating those tokens from the input that are expected next; a traditional LR parser on the other hand creates the tokens first and then tries to match them to rules from the grammar.

Due to the lack of a lexer, scannerless parsers are well suited for language composition as they remove the need for composing the lexing rules as well as the grammar rules. However, scannerless parsers introduce problems through the longest match ambiguity (i.e. it is not always clear if a longer match is preferable over multiple smaller matches) and

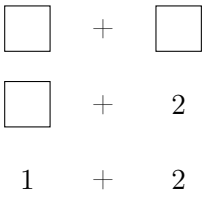


Figure 2.4: Writing a program `1+2` using syntax-directed editing. The graphic shows the results after each keypress. The user needs to start by writing '+', which creates a subtree for additions leaving two blank nodes for the left- and right-hand-side. These nodes can then be filled with the rest of the expression.

the reserved-identifier ambiguity (i.e. should keywords be preferred over identifiers) [68]. To solve these problems it is necessary to add reject rules to the grammar [85], thus ignoring other interpretations of a program and making the parser non-context-free [80] and undefined under composition.

2.3.3 PEG

PEGs (Parsing Expression Grammars) are a modern update of a classic parsing approach [32, 35]. PEGs have no relation to CFGs. They are closed under composition (unlike LR grammars) and are inherently unambiguous (unlike generalised parsing approaches). Both properties are the result of the *ordered choice* operator e_1 `/` e_2 which means "try e_1 first; if it succeeds, the ordered choice immediately succeeds and completes. If and only if e_1 fails should e_2 be tried." However, this operator means that simple compositions such as `S: S1 / S2`, where S1 and S2 are the grammars of two languages with `S1: 'a'` and `S2: 'a' 'b'`, fail to work as expected, because if the LHS matches `a`, the RHS is never tried, even if it could have matched the full input sequence. This means one has to be very careful when composing grammars together so as to not accidentally 'hide' one of the grammars due to the other one always succeeding. To make matters worse, in general such problems cannot be determined statically, and only manifest when inputs parse in unexpected ways.

2.3.4 Syntax-directed editing

Syntax-directed editing (SDE) works very differently to traditional parsing approaches, always operating on an abstract syntax tree (AST). AST elements are instantiated as templates with holes, which are then filled in by the user (see Figure 2.4).

This means that programs being edited are always syntactically valid and unambiguous (though there may be holes with information yet to be filled in). This side-steps the

flaws of parsing-based approaches, but because such tools require constant interaction with the user to instantiate and move between AST elements, the SDE systems of the 70s and 80s (e.g. [74, 4, 10, 21, 60]) were rejected by programmers as restrictive and clumsy [42, 48, 57] because they didn't align with the way programmers reason about writing code [73, 50].

More recently, the MPS editor has relaxed the SDE idiom, making the entering of text somewhat more akin to a normal text editor [64]. In essence, small tree rewritings are continually performed as the user types, so that typing 2, Space, +, Space, 3 transparently rewrites the '`2`' node to be the LHS of the '`+`' node before placing the cursor in the empty RHS box of the '`+`' node where '`3`' can then be entered in. This lowers, though doesn't remove, one of the barriers which caused earlier SDEs to disappear from view. Authors have to manually specify such rewritings for each language. Furthermore, the rewritings only affect the entry of new text. Editing a program still feels very different from a normal text editor. For example, deleting nodes requires great care and special actions, e.g. deleting '`+ 2`' from the expression '`1 + 2 * 3`' is not possible without also deleting '`*`', which then has to be re-entered. Similarly, only whole nodes can be selected from the AST, so one cannot copy '`2 +`' from the expression '`2 + 3`' on-screen.

Put another way, MPS is sometimes able to hide that it is an SDE tool, but never for very long. The initial learning curve is therefore relatively steep and unpleasant for many programmers.

2.3.5 Summary

In summary, when it comes to language composition, parsing approaches are either too limited (LR parsing), allow ambiguity (generalised parsing), or are hard to reason about (PEG parsing). While approaches such as Copper [82, 70] and Spoofax [41] have nonetheless been used for some impressive real-world examples, I believe that such issues might limit uptake. Syntax-directed editors on the other hand are ideal for language composition as there is no need for parsing and thus no ambiguity. Unfortunately, SDEs are difficult to learn and get used to and many programmers have rejected them in the past due to their steep learning curve and unfamiliar work flow.

2.4 An overview of incremental lexing and parsing

This section gives an overview of incremental lexing and incremental parsing which will be explained in more detail in Sections 2.5 and 2.6.

```
// Parser
Var: Id "eq" Val;
Id: "id";
Val: "int";

// Lexer
eq:"="
id:"[a-z]+"
int:"[0-9]+"
```

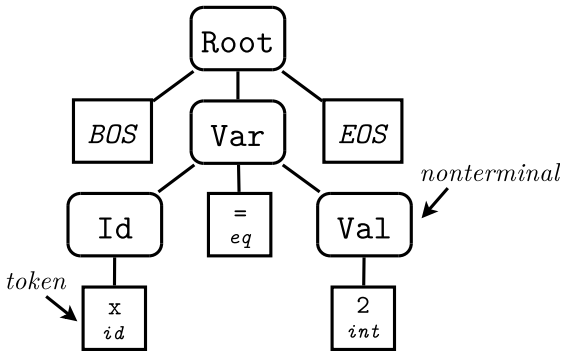


Figure 2.5: An example of a parse tree constructed from the grammar on the left. The minimal parse tree consists of three special nodes: a *Root* nonterminal; and *BOS* (Beginning of Stream) and *EOS* (End of Stream) terminals (both children of *Root*). For brevity reasons, these nodes are elided for the remainder of this thesis. All nodes created from user input are (directly or indirectly) children of *Root* and are contained between *BOS* and *EOS*.

Traditional parsing is a batch process: an entire file is fed through a parser and a parse tree is created from it. Incremental parsing, in contrast, is an online (i.e. interactive) process: it parses text as the user types and continually updates a parse tree. In other words, incremental parsing allows an editor to re-parse a program after every keypress, resulting in an always up-to-date parse tree, which can then be used for semantic analysis or syntax highlighting. A number of incremental parsing algorithms were published from the late 70s [33, 37, 49, 65] to the late 90s, gradually improving efficiency and flexibility [53, 26]. The last major work in this area was done in 1998 by Wagner [88] who defined a number of incremental parsing algorithms. The parser used in this thesis is based on his LR-based incremental parser which has two major benefits: it handles the full class of LR(k) grammars; and has formal guarantees that the algorithm is optimal (i.e. the number of steps needed to integrate user changes into an existing parse tree is minimal). Furthermore, Wagner presents an algorithm for incremental lexical analysis, improving on works such as [29, 4, 5, 27]. The incremental lexer in this thesis is based on Wagner's, although diverges from his implementation in some of its finer details. This section gives a brief overview of incremental lexing and parsing, while Sections 2.5 and 2.6 explain them in more detail.

An incremental parser and lexer both operate on a parse tree. Parse tree nodes are either *nonterminals* (representing rules in the grammar) or *tokens* (representing terminal symbols). Nonterminal nodes are immutable (i.e. their type cannot change) and have zero or more ordered child nodes (which are not immutable and can be updated). Tokens have a type (e.g. 'int') and a value (e.g. '3') and are mutable. Figure 2.5 shows an example of a parse tree and Figure 2.6 gives an overview of the attributes and methods of a parse tree node.

`value: String`
: The text value of the token represented by this node. None if the node is a nonterminal.

`symbol: Symbol`
: The lookup symbol of this node. Returns the rule name if the node is a nonterminal, or the terminal type if the node is a token.

`state: int`
: The state in which this node was pushed onto the parse stack.

`changed: bool`
: True if this node was changed since the last re-parse.

`nested_changes: bool`
: True if there are unparsed changes within this node's subtree.

`exists: bool`
: Determines whether this node exists in the current version of the tree.

`__len__() -> int`
: Overloaded function used by `len`. Returns the character length of this node's value.

`left_sibling(version: int) -> Node`
: This node's left sibling in the specified version. Refers to the current version if no version is specified. Returns None if the node is the last child.

`right_sibling(version: int) -> Node`
: Like `left_sibling`, but returns None if the node is the first child.

`previous_token() -> Node`
: Returns the earliest node representing a token that precedes this node.

`next_token() -> Node`
: Returns the earliest node representing a token that succeeding this node.

`update(t: Token)`
: Update this node's token information (i.e. value, lookahead, lookback) using the given token.

`text_length(version: int) -> int`
: Returns this node's textual yield. If the node is a nonterminal this is the sum of the text-length of all its children.

Figure 2.6: Overview of some of the attributes and methods of a parse tree node. The `version` argument refers to a specific version of the parse tree as described in Section 2.6.1.

When the user types, the incremental lexer first either creates, or updates, tokens in the parse tree. The lexer considers where the cursor is in the tree (i.e. where the user is typing) and uses lookahead knowledge stored in the nodes representing those tokens to work out the affected area of the change. Newly created tokens are then merged back into the tree. In the simple case where a token's value, but not its type, was changed, no further action is needed. In all other cases, the path from each changed token up to the root is first marked with *nested change* flags which are then used by the incremental parser to update the parse tree. The incremental parser starts at the beginning of the tree and tries to re-parse all subtrees that contain changes. Assuming the user's input is syntactically valid, nonterminals are created or removed, as appropriate. Unchanged subtrees are reused as is whenever possible. Since nonterminals are immutable, subtrees which can't be reused must be recreated from scratch.

Syntactically incomplete programs lead to temporarily incorrect parse trees. In such cases, the incremental parser typically attaches new tokens to a single parent. When the user eventually creates a syntactically valid program, the tree is rewritten.

2.5 Incremental Lexing

An incremental lexer can re-lex user edits and integrate the result into the current stream of tokens, without having to lex the entire input from scratch. While the performance gain is negligible, incremental lexing is a requirement for incremental parsing, as it allows the parser to distinguish between old and new parts of a program so that it can only re-parse those parts that have been added or altered. Wagner explains incremental lexing in [88, p. 37]. However, he doesn't show how tokens, once re-lexed, can be integrated back into the parse tree. Sections 2.5.2, 2.5.3, and 2.5.4 summarise his explanations, while Sections 2.5.5 and 2.5.6 show my own solution for merging re-lexed tokens back into the parse tree. Section 2.5.7 then shows an optimisation for the merging algorithm from 2.5.6 that minimises the amount of new nodes that need to be created by cleverly overwriting existing nodes within the parse tree.

2.5.1 The problem with traditional lexing

The input of a traditional lexer is a stream of characters. The lexer reads one character at a time from the input and generates tokens from it. Tokens consist of *(value, type)* pairs and are defined by lexing rules which consist of a name and a regular expression. For instance, the lexing rules of a grammar that allows basic arithmetic could look like this:

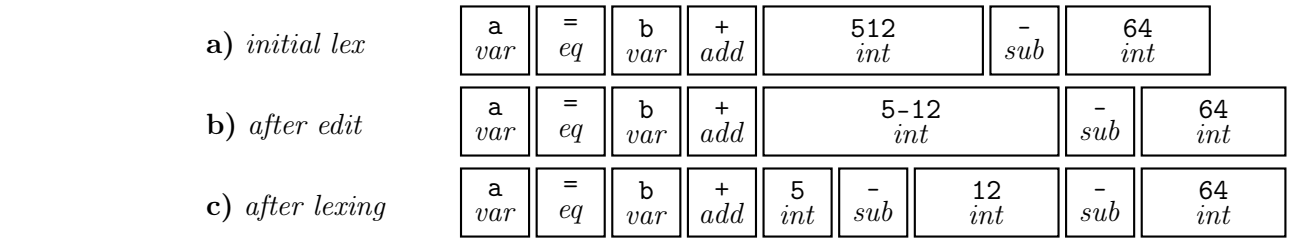


Figure 2.7: An example showing that re-lexing changes to a program typically results in mostly the same tokens. **a)** Token sequence of the program "`a=b+512-64`" when initially lexed. **b)** The user edited the token `512` by inserting '`-`' in-between '`5`' and '`12`'. **c)** The new token sequence after re-lexing the user change. Even though the whole whole program has been re-lexed, it results in mostly the same tokens as before, with only token '`512`' being split into two tokens '`5`' and '`12`' and a new token '`-`' inserted in-between.

```
add : '+'
sub : '-'
int : [0-9]+
ws  : [ \n]+
var : [a-zA-Z][a-zA-Z0-9_]*
```

When we generate a lexer from these rules and use it on the input "`32 + 512`" the lexer returns the following sequence of tokens:

```
('32', int) (' ', ws) ('+', add) (' ', ws) ('512', int)
```

When a user edits a program, most of their changes will be comparatively small given the overall size of the source code. However, a traditional lexer will always re-lex the whole program, even if most of it remains unchanged (see Figure 2.7 for an example).

2.5.2 Outline of an incremental lexer

The goal of an incremental lexer is to only re-lex those parts of a program that have been changed. In an editor based on an incremental parser, edits to a program are directly translated to the nodes in the parse tree representing tokens. For example, when the user edits the token '`512`', by inserting a '`-`' in-between the digits '`5`' and '`12`', the value of the node in the parse tree representing that token is updated to `5-12` and the node is marked as *changed*. The incremental lexer then searches for all nodes that were marked as changed, re-lexes them with a traditional lexer, and then merges the results back into the tree. In other words, an incremental lexer detects which nodes need to be re-lexed and handles the merging of the results back into the parse tree, while the creation of tokens from the input is delegated to a normal lexer. In the remainder of this thesis we thus use the term lexer to describe the tokenisation of user input, and the term incremental lexer when referring to the algorithms for the detection and merging of changes.

```
// Lexing rules
cls: "class"
as: "as"
chr: "[a-z]"
```

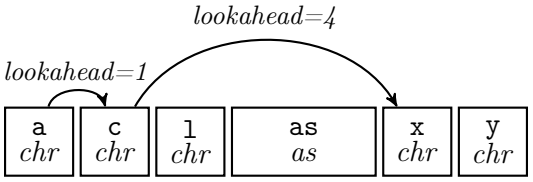


Figure 2.8: An example for token lookahead showing some lexing rules on the left, and the resulting tokens, when lexing the input '`aclasxy`', on the right. The token '`a`' has a lookahead of 1. This is because, the lexer had to read one additional character to make sure that token doesn't also match the type *as*. However, since the following character is '`c`', the lexer produced the token '`a`' with type *chr*. The next token, '`c`' has a lookahead of 4, because the lexer had to read all the way up to '`x`', before it could decide that it wasn't possible to lex the longer match '`class`' of type *cls*. All other tokens have lookaheads of 0.

When a token is changed, it is not uncommon that this change affects other tokens before and after the change, which then also have to be re-lexed. We call such tokens *dependent* tokens. To re-lex a changed token appropriately, we first have to find the furthest token *before* the changed token, that is dependent on it. The node that represents that token is where re-lexing starts. From here, the incremental lexer starts producing new tokens from processing nodes in the parse tree. Re-lexing only stops, if a token was re-lexed to itself (i.e. the new token's value and type match exactly the node that it was produced from). Afterwards, the new tokens are merged back into the parse tree, overwriting all nodes that were processed during re-lexing. To compute dependent tokens we use *lookahead* and *lookback* values stored within each token, which is explained in the following two subsections. The remainder of this section gives some examples of the re-lexing process and shows an algorithm for optimally merging tokens back into the parse tree.

2.5.3 Token lookahead

The lookahead of a token is the number of characters following that token that the lexer had to inspect before deciding that the token is complete. Lexers typically try to find the longest match when tokenising a program. To verify that a generated token is indeed using all available characters matching its type, the lexer needs to scan characters exceeding the length of the token. For example, a lexer that knows about integer values (e.g. `[0-9]+`) that is given the program '`512+64`', would produce '`512`' as the first token. However, to confirm that '`512`' is the longest match, it will have to scan '`+`' as well. But since there is no lexing rule describing a token '`512+`', the lexer returns '`512`' with a lookahead of 1. In most programming languages, tokens typically have a lookahead of either 0 (e.g. brackets) or 1 (e.g. identifiers). However, larger lookaheads are possible as the example in Figure 2.8 shows.

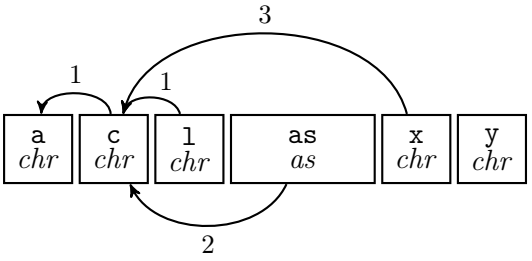


Figure 2.9: Lookback values for the example in Figure 2.8. The token '`c`' has a lookback of 1, because it can be reached by '`a`'. This means that changing '`c`' would result in '`a`' being re-lexed as well. Since the lookahead of '`c`' reaches all the way to '`x`', the lookback values of tokens '`l`', '`as`', and '`x`' all point back to '`c`'. This means that changing any of them, would require re-lexing to start from '`c`'.

Using lookahead values we can determine where re-lexing has to start after a change, by scanning all tokens before that change and checking if their lookahead value reaches the changed token. In other words, if the lookahead value of a token a is greater than the character distance between a and another token b, then a depends on b and needs to be re-lexed when b is changed. For example, if we assume that in Figure 2.8 the user changes the '`x`' into an '`s`', then instead of just re-lexing '`x`', we need to start re-lexing at token '`c`', since its lookahead reaches '`x`', resulting in a new token '`class`'.

Unfortunately, lookahead values on their own are too inefficient to find all dependent tokens of a change. Lookahead can only tell us if a token a depends on a token b. However, when changing a token b we need to find the furthest token a that depends on b. In order to make sure that there are no further dependent tokens, we need to scan all tokens from b up to the beginning of the program. Instead, we need a method that, given a changed token b, tells us how far back we need to go, without scanning the entire file in the worst case scenario. This can be achieved by converting lookahead values into lookback counts.

2.5.4 Lookback counts

The lookback value of a token b defines its distance in tokens to the furthest token a that depends on b. It is used to determine how far back re-lexing needs to start when a token was changed by the user. It can be calculated and updated incrementally from the lookahead values of the re-lexed tokens, immediately after they have been merged back into the parse tree. The algorithm is shown in Listing 2.2. Figure 2.9 shows the calculated lookback values for the example from Figure 2.8.

The algorithm starts with the token where re-lexing was started as input and iterates over the following tokens while assigning lookback values to them. The currently processed token is `n`. The variable `la_list` stores (*lookahead*, *count*)-tuples for each token that

```
1  def update_lookback_W(n: Node):
2    la_list: List[int, int] = []
3    while type(n) is not EOS:
4      la_list = remove(la_list)
5
6      newlookback: int = max_count(la_list)
7      if not was_relexed(n) and \
8            n.lookback == newlookback:
9        break
10     n.lookback = newlookback
11
12     # advance step
13     c: int = len(n.value)
14     la_list = advance(la_list, c)
15
16     # add step
17     la_list.append((n.lookahead, 1))
18
19     n = n.next_token()
```

```
20 def remove_W(la_list: List[(int, int)]):
21   new: List[int, int] = []
22   for la, count in la_list:
23     if la > 0:
24       new.append(la, count)
25   return new
26
27 def advance_W(la_list, c: int):
28   new: List[int, int] = []
29   for la, count in la_list:
30     new.append(la - c, count+1)
31   return new
32
33 def max_count_W(la_list: List[(int, int)]):
34   maxc: int = 0
35   for la, count in la_list:
36     if count > maxc:
37       maxc = count
38   return maxc
```

Listing 2.2: A Python version (with type annotations) of Wagner's algorithm for calculating lookback counts from lookahead values. The algorithm is run after re-lexing has finished and takes as input the token where re-lexing was started. It then iterates over all re-lexed tokens and updates their lookback values. The algorithm stops if it encounters a token that wasn't re-lexed and whose lookback value hasn't changed. Note that, since tokens are represented by nodes in the parse tree, the input of this function is a node.

was processed, where the continuously updated *count* holds the distance in tokens that the lookahead value spans. For each new token that is processed, its lookahead value is added to `la_list`, and *count* is initialised with 1 (line 17). Each entry in `la_list` represents a preceding token that has lookahead into `n`. Each time another token is processed, its text-length is subtracted from all entries in `la_list` and *count* is increased by 1 (line 27–31). If an entry's lookahead reaches 0, it is removed from the list (lines 20–25). This way `la_list` only ever contains entries which can reach the current token `n`. The maximum `count` value within `la_list` defines the furthest preceding token from `n` that has lookahead into `n`, and thus defines `n`'s lookback value (line 10). The algorithm updates the lookback values of tokens, until it processes a token that wasn't previously re-lexed and whose lookback value remains unchanged (lines 7–9).

To help the understanding of the algorithm, the following applies it to the example from Figure 2.8, assuming all tokens have just been re-lexed. In this example `la_list` also stores the token values to show which token each entry belongs to, though this is not necessary in the actual algorithm. We start the function with token '`a`' as an argument. Since `la_list` is empty the *remove* step (line 4) does nothing and `newlookback` (and thus `a`'s lookback) is set to 0. `la_list` is still empty, so the *advance* step (lines 13–14) also has nothing to do. Next we add the current token's lookahead value to `la_list` and initialise `count` with 1 (line 17). The following shows the elements of `la_list` in form of a table, though in the algorithm the data structure is a list of tuples:

Token	*Lookahead*	*Count*
`a`	1	1

The next processed token is '`c`'. Since none of the entries' lookahead value is 0, no entry is removed from `la_list` (line 4). However, `newlookback` is set to the maximum `count` value, which is 1 (line 6). At this point we check if we are already done, which would be the case if the current token hasn't been previously re-lexed and its lookback value already matched the calculated one (lines 7–9). But since '`c`' was re-lexed, we continue and set its lookback to 1 (line 10). Now we advance the list by increasing all `count` values by 1 and reducing all lookahead values by the current token's text-length (lines 13–14). Afterwards we add the current token's lookahead value to the list (line 17), resulting in:

Token	*Lookahead*	*Count*
`a`	0	2
`c`	4	1

In the next iteration we process token '`l`'. The *remove* step removes token '`a`' from the list since its lookahead is now 0 which means it doesn't reach any of the following tokens. Again, we assign the maximum `count` value, 1, as lookback to token '`l`'. After advancing `la_list` and adding `l`'s lookahead, we get:

Token	*Lookahead*	*Count*
`c`	3	2
`l`	0	1

We now process '`as`'. In the *remove* step we immediately remove '`l`' again and assign lookback value 2 to '`as`'. After advancing and adding, `la_list` changes to:

Token	*Lookahead*	*Count*
`c`	1	3
`as`	0	1

Next we process '`x`'. First we remove '`as`' from the list during the *remove* step. Then we assign a lookback value of 3 to `x`. After advancing and adding we get:

```
1  def inclex_Wb(n: Node):
2    start: Node = find_relex_start(n)
3    temp: Node = start.previous_token()
4    processed: List[Node] = []
5    generated: List[Token] = []
6    lenp = leng = 0
7    while True:
8      old: Node, new: Token = lexer.next_result()
9      if n has been re-lexed:
10       if lenp == leng and old == new:
11         break
12     processed.append(old)
13     generated.append(new)
14     lenp += len(old)
15     leng += len(new)
16   merge_back(iter(processed), iter(generated))
17   update_lookback(temp.next_token())
```

```
18 def find_relex_start_Wb(n: Node):
19   for i in range(n.lookback):
20     n = n.previous_token()
21   return n
```

Listing 2.3: Simplified algorithm for incremental lexing. The function `inclex` is called on the token that was edited. We start by using its lookahead to find the furthest affected token, where lexing needs to begin (line 2). The lexer then generates new tokens from the nodes in the parse tree and stores them, along with the nodes that were processed during lexing, in two separate lists (lines 12-13). Lexing stops when a node is re-lexed to itself, i.e. the new token has the same type and value as the node it was produced from and their offsets align (Line 10). Afterwards, the new tokens are merged back into the parse tree (line 16). To update lookbacks we need to store the node before re-lexing started (line 3), since the starting node may have been replaced during the merge.

Token	*Lookahead*	*Count*
`c`	0	4
`x`	0	1

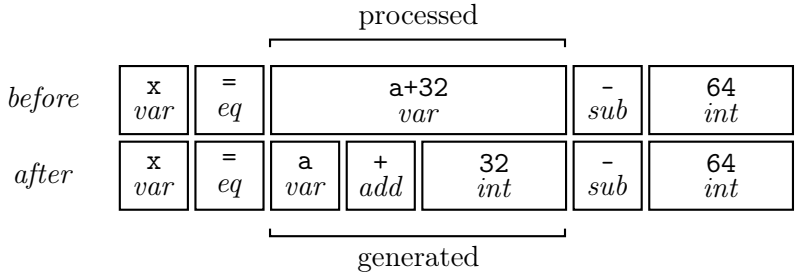
Finally, we reach the token `y`. In the *remove* step we remove both '`c`' and '`x`', resulting in an empty `la_list` and `newlookback` being 0. Assuming that the token '`y`' wasn't re-lexed in the previous edit and its current lookback value is already 0 the algorithm would end here. If, however, '`y`' has either been re-lexed or its lookback doesn't match `newlookback`, the algorithm would have to continue updating the lookbacks of any tokens following '`y`', until this check succeeds or reaches the end of the program.

2.5.5 Re-lexing all affected tokens

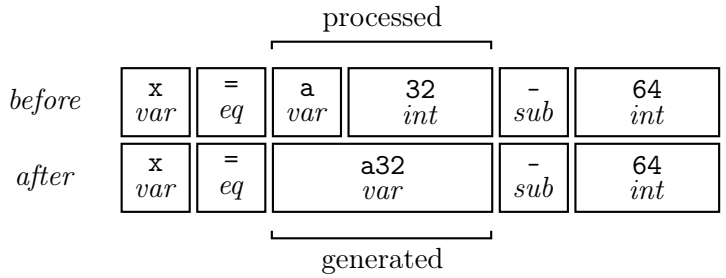
Using the lookback values we can now easily find all preceding nodes that need to be included in the re-lexing phase. After finding the furthest preceding node that is affected, the incremental lexer starts re-lexing at that node's position. As a minimum, the lexer re-lexes all nodes up to and including the edited token. However, it may continue re-lexing as long as this leads to new tokens being generated. It only stops, if it encounters a node that doesn't change after being re-lexed. A simplified algorithm for incremental lexing,

based on Wagner's description, is shown in Listing 2.3. The incremental lexer differs from Wagner's, in that during re-lexing it builds two lists, one storing all nodes that were *processed*, and one storing all tokens that were *generated*. These are later used to merge the generated tokens back into the parse tree.

Most edits, when being re-lexed, lead to one of two scenarios: either more tokens have been generated than nodes were processed, or more nodes have been processed than tokens were generated. The former typically occurs when a node is split into two or more tokens. The latter happens when two (or more) nodes are combined to a single token. A third, trivial scenario is the editing of a token's value without causing a type change, which does not require any further action. The following example shows some input before and after re-lexing, where the user changed the token '`a32`' into '`a+32`', resulting in the splitting up of the node representing that token:



When the token '`a32`' is edited, re-lexing starts at its position, since its lookback value is 0. Re-lexing stops as soon as the incremental lexer processes '`-`', which is re-lexed to itself, and thus means that from here on re-lexing doesn't lead to any more changes. The next example shows the inverse operation of the previous example, where the user deletes the '`+`' again, resulting in the merging of two nodes:



When a token is deleted, the incremental lexer uses its left neighbour to initialise re-lexing. Here, deleting '`+`' means re-lexing starts with '`a`'. Since `a`'s lookback is 0, no further tokens are included in the re-lexing phase. Lexing stops at '`-`', as re-lexing it results in the same type as before.

```
1  def merge_back_D(processed: Iter[Node], generated: Iter[Token]):
2    # insert new nodes into tree
3    for new: Token in generated:
4      old: Node = processed.next()
5      if old is not None:
6        old.update(new) # overwrite value/type/etc with new token data
7        last = old
8      else:
9        # insert remaining generated tokens into parse tree
10       insert Node(new) immediately after last
11       last = new
12
13   # delete left over processed nodes from parse tree
14   while True:
15     old: Node = processed.next()
16     if old is not None:
17       remove(old)
18     else:
19       break
```

Listing 2.4: Merging re-lexed tokens back into the parse tree. The function takes iterators over the processed nodes and generated tokens as input. It then iterates over all generated tokens and uses them to overwrite the processed nodes with (lines 3–7). If there are no processed nodes left to overwrite, the remaining generated tokens are inserted afterwards (lines 8–11). If there are no generated tokens left, all remaining processed nodes are deleted (lines 13–18). For simplicity we assume that iterators return `None` instead of raising an exception after the last element has been returned.

2.5.6 Updating the token sequence

Although it is not explicitly explained, Wagner's description suggests that his algorithm simply removes re-lexed nodes and replaces them with the results from the lexer, and then relies on top-down reuse (see Section 2.8.2) to recover and reuse old nodes. The following presents a, to the best of my knowledge, novel approach of merging re-lexed tokens back into the parse tree, that can be used to optimise node reuse during re-lexing.

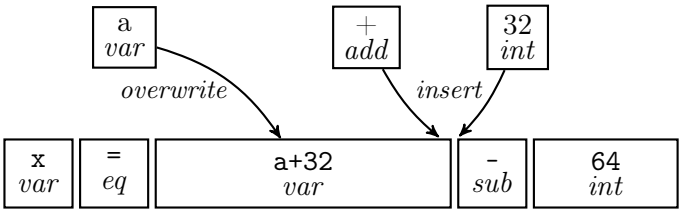
After the incremental lexer has re-lexed all nodes, it has produced two lists: one containing all processed nodes from the parse tree; and another containing newly generated tokens. In the next step, the newly generated tokens need to be inserted into the parse tree, replacing all processed nodes in the process. However, instead of removing all old nodes and then inserting the new tokens in their place, it is more memory efficient to overwrite existing nodes first and then remove any remaining nodes or insert any remaining tokens. Listing 2.4, shows an algorithm that updates a token sequence by overwriting processed nodes with the newly generated tokens. The algorithm can be split up into three scenarios which are described as follows.

Only a single token was affected

In many cases, when a program is edited, the change only affects a single token. For example, if only the value of a token is updated, e.g. changing a number '`2`' to '`23`', no other token need to be re-lexed. Also, since the type of the token remains the same, it is not necessary to re-parse the program. However, if the type of the token was changed, e.g. changing '`2`' to '`a2`' may update its type from *int* to *var*, a re-parse of the change is required.

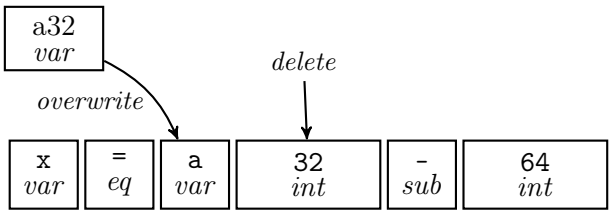
Inserting tokens

After a node has been split into multiple tokens, in order to update the token sequence, the algorithm from Listing 2.4 overwrites the processed nodes one by one with the generated tokens. If at the end there are no more nodes left to overwrite, new tokens are inserted. In the first example from Section 2.5.5, where a token '`a+32`' was re-lexed into 3 separate tokens, we thus first overwrite '`a+32`' with '`a`'. Since that was the only processed node, there are no more nodes left to overwrite, so the remaining tokens '`+`' and '`32`' are inserted after '`a`', as illustrated below:



Removing tokens

When multiple nodes are combined into a single token, the token sequence is updated by overwriting all processed nodes with the generated tokens. If all generated tokens have been merged this way, the remaining processed nodes are removed from the parse tree. In the second example from Section 2.5.5, where the two tokens '`a`' and '`32`' were merged, we first overwrite the old node '`a`' with '`a32`'. Afterwards, the remaining processed node, '`32`', is deleted:



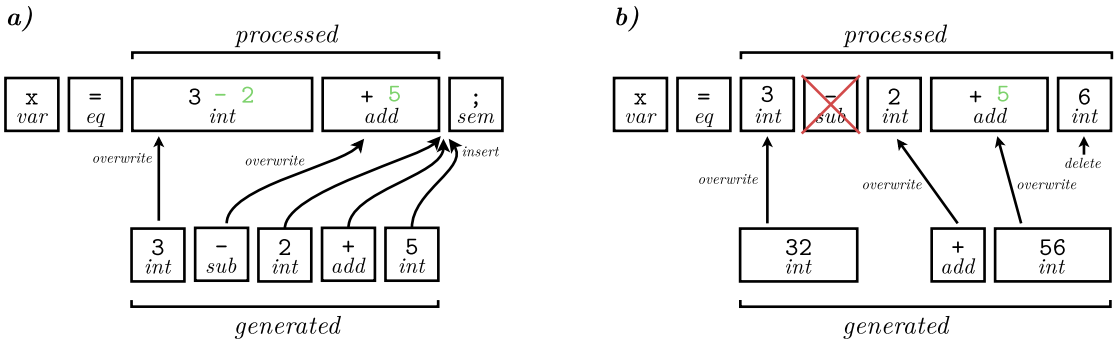


Figure 2.10: Two examples showing how the simple merge algorithm from Listing 2.4 integrates re-lexed tokens back into the parse tree. Since the algorithm doesn't take into account from which old node a new token was generated, it sometimes merges new tokens with a different node than they were created from. This results in more type changes than necessary and may invalidate any annotations stored on that node. **a)** The user has made two edits (shown in green) which were re-lexed in one go. We can see that the node '`+5`' (formerly '`+`') is overwritten with '`-`'. A new '`+`' token is then later inserted at the end of the merge process. **b)** The user has made a deletion and an insertion, which leads to '`+5`' (formerly '`+`') being overwritten by '`56`' forcing it to update its type. If it were to be overwritten by '`+`' only the value would need changing.

2.5.7 Improving the merging algorithm

The algorithm shown in Listing 2.4 has a small downside: since it doesn't know which new tokens were produced from which processed nodes, it sometimes overwrites nodes in a suboptimal order. The problem is illustrated in Figure 2.10.

We can minimise the amount of node changes by splitting up the generated tokens and processed nodes into groups and merging each group separately. Each group matches a set of generated tokens with the processed nodes they were constructed from. This way we avoid that generated tokens in one group can overwrite processed nodes in the other, which leads to more opportunities of merging a generated token into the same node as before. Figure 2.11 shows how the examples from Figure 2.10 are integrated when using an improved merging algorithm.

The improved merging algorithm is shown in Listing 2.5. It introduces two new variables, `oldlen` and `newlen`, which store the amount of characters that have been processed so far, for processed nodes and generated tokens. A new group begins whenever the two variables are equal. This allows it to align generated tokens more often with their old counterpart, which reduces the amount of changes that require re-parsing. The algorithm receives the (ungrouped) generated tokens and processed nodes as input. By monitoring the total text-length of the nodes being processed, it can determine the range of each group and use this information to decide whether a token needs to be inserted, or nodes need to be removed or overwritten.

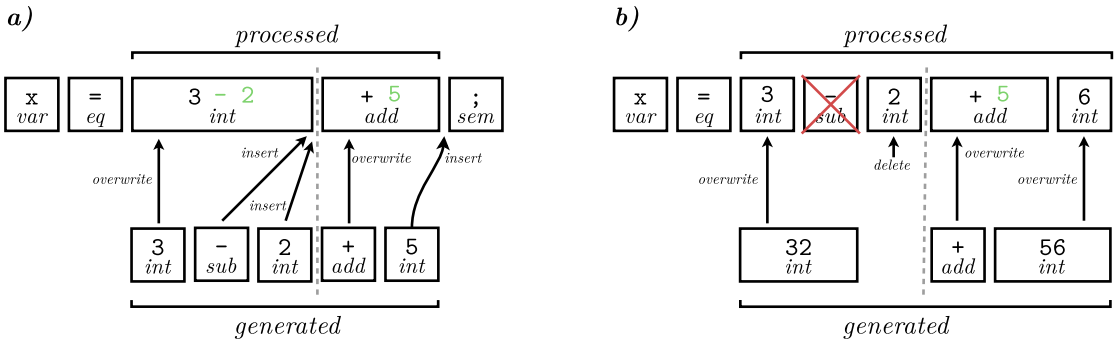


Figure 2.11: An example showing an improved merging algorithm that splits up generated tokens into groups and merge them separately. **a)** Since the merging of '3-2' was done separately, we can overwrite '+5' (formerly '+') with '+', reverting it back to its old value from before the edit. **b)** Whereas in Figure 2.10 '+5' was overwritten with '56', and '6' was deleted, the improved version allows us to overwrite both, and only having to update their values.

```
1  def merge_back_D(generated: Iter[Token], processed: Iter[Node]):
2    old: Node = processed.next() # returns the next processed node or None
3    new: Token = generated.next()
4    newlen = oldlen = 0
5
6    while old or new:
7      if oldlen >= newlen + len(new):
8        # Insert new token
9        insert Node(new) immediately after last
10       last = new
11       newlen += len(new)
12       new = generated.next()
13     elif oldlen + len(old) <= newlen:
14       # Remove processed node
15       remove(old)
16       oldlen += len(old)
17       old = processed.next()
18     else:
19       # Overwrite old node
20       oldlen += len(old)
21       newlen += len(new)
22       old.update(new)
23       last = old
24       new = generated.next()
25       old = processed.next()
```

Listing 2.5: Improved algorithm for merging re-lexed tokens, which divides generated tokens into groups. This reduces the amount of changes necessary, when integrating newly generated tokens into the parse tree. When a node is re-lexed into multiple new tokens, the algorithm first overwrites the old node with the first generated token (lines 19–25), and then immediately inserts the remaining tokens afterwards (lines 7–12). Similarly, if multiple nodes were combined into a single token, the algorithm overwrites the first of the nodes with the new token (lines 19–25), and then immediately deletes any excess nodes from the parse tree (lines 13–17).

2.6 Incremental parsing

In order to create an incremental parser, we first take a traditional parser and modify it, so that it takes as input a parse tree with user changes. From this input the incremental parser then generates a new parse tree by reordering the changes within the old parse tree according to the grammar. The parser is then optimised, so that it can reuse unchanged subtrees from the previous parse tree without having to re-parse them. This reduces the amount of work the parser has to do on each re-parse which greatly improves its performance. Sections 2.6.1 to 2.6.3 summarise Wagner's descriptions for creating an optimal incremental parser. Section 2.7 discusses the versioning of parse trees and fills in some details which Wagner only touches upon. Section 2.8 explains another of Wagner's optimisations, node reuse, and discusses two minor problems in one of his algorithms and how they can be fixed. Section 2.9 evaluates incremental parsing performance compared to traditional parsing.

2.6.1 Operating on parse trees

When changing a parser to take a parse tree as input, the process of parsing that input is, at first, fairly similar to that of a traditional parser. The only difference is that, instead of a stream of tokens, an incremental parser processes a tree of nonterminals and tokens. An (unoptimised) incremental parser then simply extracts all terminal symbols from the tokens in the parse tree and processes them just like a traditional parser would. We can do this by simply traversing the parse tree in a depth-first order. Nonterminals are processed by breaking them down and processing their children. Tokens are parsed as in a traditional parser. When the parser reaches an accept state after all tokens have been parsed, a new parse tree can be found on top of the stack. When the parser reaches an error state, error recovery is invoked which is discussed in Chapter 3. Listing 2.6 shows a simplified version of Wagner's parsing algorithm that takes a parse tree as its input[1].

Following Wagner's terminology, we call the new parse tree being constructed the *current version*, while the parse tree that was used as input is referred to as the *previous version*. At the moment these two parse trees are, apart from their tokens, completely separate. However, with optimised incremental parsing (Section 2.6.2) and node reuse (Section 2.8) the previous and current version of the parse tree may share entire subtrees between each other.

[1]Note that in this version parsing states are stored on the nodes that are pushed onto the parsing stack, instead of the stack itself. While this doesn't have any performance benefits, storing states on nodes slightly simplifies the code examples and makes life a bit easier later on in Section 3.4

```
1  state: int = 0
2  stack: List[Node] = []
3
4  def incparse_W(bos: Node):
5    la: Node = next_lookahead(bos)
6    while True:
7      if la is a terminal:
8        action = parsetable.lookup(state,
9                                   la.symbol)
10       if action is Shift:
11         shift(la, action)
12         la = next_lookahead(la)
13       elif action is Reduce:
14         reduce(action.production)
15       elif action is Accept:
16         return True
17       elif action is Error:
18         la = recover() # error recovery
19     else: # la is a nonterminal
20       la = left_breakdown(la)
```

```
21 def left_breakdown_W(la: Node) -> Node:
22   if len(la.children) > 0:
23     return la.children[0]
24   else:
25     return self.next_lookahead(la)
26
27 def next_lookahead_W(la: Node) -> Node:
28   while la.right_sibling(prev) is None:
29     la = la.get_parent(prev)
30   return la.right_sibling(prev)
31
32 def shift_W(la: Node, s: Shift):
33   stack.append(la)
34   state = la.state = s.state
35
36 def reduce_Wb(p: Production):
37   children: List[Node] = []
38   for i in p.length():
39     children.append(stack.pop())
40   state = stack[-1].state
41   n = Node(p.symbol, children)
42   goto = parsetable.lookup(state,
43                            p.symbol)
44   state = goto.state
45   stack.append(n)
```

Listing 2.6: Simplified Python version of Wagner's implementation of an LR(1) parser using a parse tree as input. The function `incparse` traverses the parse tree, parsing all tokens until the input is either accepted or rejected. The function `next_lookahead` is used to retrieve the next lookahead node from the previous parse tree (lines 27–30). If that node is a nonterminal, its subtree is broken down using `left_breakdown` to retrieve its leftmost token (lines 21–25). If the lookahead is a terminal symbol, the next parse action is looked up in the parse table (line 8). Afterwards, `next_lookahead` is used again to get the next lookahead to be processed. The function `next_lookahead` uses the global variable `prev` to reference the previous version of the parse tree. Though not strictly necessary at this stage, this is needed later to receive the correct lookaheads once subtrees are being reused as described in Section 2.8.

2.6.2 Optimised incremental parsing

When the parser re-parses a parse tree containing user changes, the resulting new parse tree is often very similar to the previous version of the parse tree, with many subtrees being identical (see Figure 2.12 for an example). In fact, in practice most user changes only affect a fraction of the entire parse tree and most subtrees stay the same after a re-parse.

An optimised incremental parser attempts to reuse as many subtrees as possible during a re-parse. In general, a subtree can be reused if it does not contain any changes. When the user edits a parse tree, all nodes leading from the root down to those edits are marked with a *nested changes* flag. The incremental parser can follow those markings to re-parse the user changes. If a subtree does not contain any changes, the incremental parser tries to reuse it, since re-parsing it typically results in the same subtree. Listing 2.7

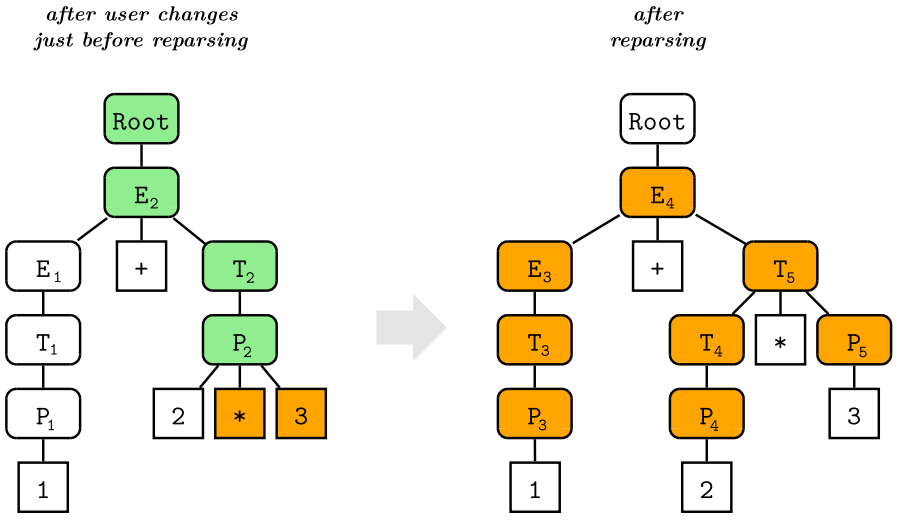


Figure 2.12: An example showing that when we re-parse changes within a parse tree, some re-parsed subtrees in the new parse tree (right) are identical to their equivalent in the previous parse tree (left). New nodes are coloured orange, nodes leading to changes are coloured green. The user inserted the new text '`*3`', which after re-lexing resulted in the parse tree on the left. After re-parsing, the subtree $\texttt{E}_1$ in the previous parse tree has been re-parsed to the exact same subtree as before ($\texttt{E}_3$).

shows a simplified version of Wagner's optimised incremental parsing algorithm. Note that this implementation requires additional information in the parse table which allows nonterminals to be used as lookups [88, p. 62]. Alternatively, we can retrieve the left-most terminal symbol within a subtree and use it as lookup to decide if a nonterminal is shiftable, though this is less efficient than using an extended parse table.

While the main requirement for a reusable subtree is that it doesn't have any changes, this does not always guarantee that it can be shifted. Sometimes changes located before the unchanged subtree changed the parse state in such a way that the unchanged subtree is not valid any more, and so it has to be broken down and re-parsed (Listing 2.7, lines 35–36). In some cases the unchanged subtree is invalid because there are still outstanding reductions on the parse stack, which need to be applied first (Listing 2.7, lines 33–34). After all reductions have been applied, the algorithm can attempt to shift the subtree again.

If a subtree can be shifted, it is done so optimistically. This means that we do not verify upfront if the subtree is truly valid, apart from checking if its nonterminal symbol is valid in the current state. Subtrees are created from reductions, which are based on the next lookahead. In other words, the reduction of a subtree is dependent on its following terminal symbol. If that symbol changes, the reduction is likely to change as well. In order to verify if a subtree can be shifted, we have to check if the next lookahead leads to the same reduction as in the last parse. Wagner's incremental parser, however, does this

```
1  state: int = 0
2  stack: List[Node] = []
3
4  def incparse_W(bos: Node):
5    verifying: bool = False
6    la: Node = next_lookahead(bos)
7    while True:
8      if la is a terminal:
9        action = parsetable.lookup(state, la.symbol)
10       if action is Shift:
11         verifying = False
12         shift(la, action)
13         la = next_lookahead(la)
14       elif action is Reduce:
15         reduce(action.production)
16       elif action is Accept:
17         return True
18       elif action is Error:
19         if verifying:
20           right_breakdown()
21           verifying = False
22         else:
23           la = recover()
24     else: # la is a nonterminal
25       if la.nested_changes:
26         la = left_breakdown(la)
27       else:
28         action = parsetable.lookup(state, la.symbol)
29         if action is Shift:
30           verifying = True
31           shift(la, action)
32           la = next_lookahead(la)
33         elif action is Reduce:
34           reduce(action.production)
35         elif action is Error:
36           la = left_breakdown(la)
```

Listing 2.7: Simplified Python version of Wagner's incremental parser, optimised to allow the reuse of subtrees. If a subtree doesn't contain changes it can be reused if the parse table determines that it is shiftable (line 28–32). In some cases it is necessary to apply some outstanding reductions first, before the subtree can be shifted (lines 33–34). Otherwise the unchanged subtree needs to be broken down (lines 35–36). When a subtree is shifted the algorithm enters a verification phase. If during an error occurs during this phase, the previously shifted subtree is broken down using `right_breakdown` (lines 18–21). The verification phase ends as soon as a terminal symbol is shifted (line 11).

```
1 def right_breakdown_W():
2   node = stack.pop() # remove optimistically shifted subtree
3   while node is nonterminal:
4     for c in node.children:
5       action = parsetable.lookup(stack[-1].state, c.symbol)
6       shift(c, action)
7     node = stack.pop()
8   action = parsetable.lookup(state[-1].state, node.symbol)
9   shift(node, action) # leave final token on stack
```

Listing 2.8: Optimistically shifted subtrees can be reverted by breaking them down to reveal their rightmost token. This can be done by removing the subtree from the parse stack and shifting all of its children back onto the stack. As long as the top of the stack is a nonterminal, this process is repeated. Since this only breaks down the rightmost child, preceding subtrees are still reused.

check lazily: reusable subtrees are shifted optimistically, and their validity is confirmed later. For this reason, whenever the parser shifts a subtree it enters a verification phase (Listing 2.7, line 30). This phase ends when the next lookahead can be shifted, which confirms the optimistic shift (Listing 2.7, line 11). If, however, an error occurs during the verification phase, then the optimistic shift was invalid and needs to be reversed and the subtree broken down. This can be done by simply removing the shifted subtree from the parse stack again and parsing its contents. However, even though the subtree couldn't be shifted, it may contain other subtrees which are still valid and which we don't want to re-parse again. We thus only have to break down the subtree to its rightmost token, since its terminal symbol confirms any other shifts within the subtree (see Figure 2.13 for an example). In Wagner's incremental parser this is done by the function `right_breakdown`, which is shown in Listing 2.8.

2.6.3 Whitespace

In most programming languages, whitespace is only important inasmuch as it separates other tokens. Traditional lexers therefore consume and discard whitespace. For an editor based on an incremental parser this is unacceptable, as we need to maintain whitespace in the parse tree to accurately render the user's input (see Section 5.2.3). A solution for whitespace handling is suggested by Wagner [88, p. 129], which we can adopt with minor variations.

When a grammar is configured to have implicit whitespace[2], the grammar is automatically mutated such that references to a production rule `WS` are inserted before the first, and after every, terminal in the grammar. Although the user can define `WS` to whatever they want, a common example of what is added to the grammar and lexer is as follows:

[2]In *Eco* this can be achieved by setting the `%implicit_whitespace=true` flag within the grammar.

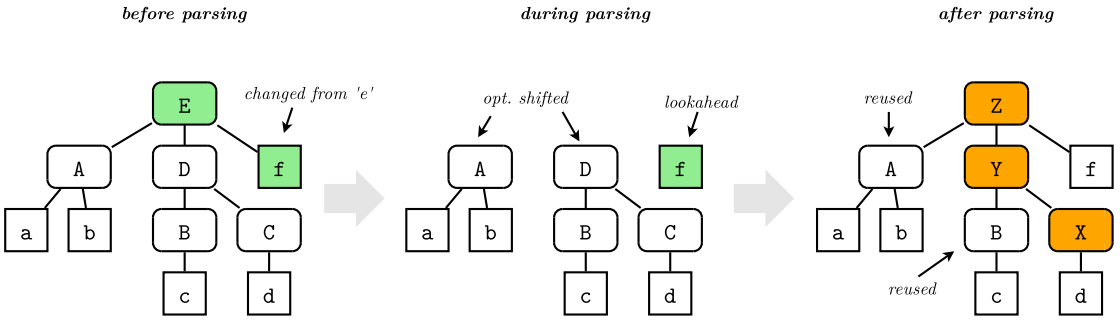


Figure 2.13: An example for an optimistically shifted subtree, which needs to be reverted because it couldn't be verified. Before parsing, the user changed a token '`e`' to '`f`', which causes the parse tree to be re-parsed. During re-parsing the subtrees `A` and `D` can be optimistically shifted. However, subtree `D`'s previous reduction depended on lookahead '`e`' which has been changed. During the verification phase, the changed token `f` results in an error. This means that subtree `D` needs to be broken down. However, it contains another subtree `B`, which has no changes and whose reduction is already confirmed by '`d`'. We thus only need to break down `D` to the rightmost token. After re-parsing, the changed token '`f`' has caused some subtrees to be re-parsed, while `A` and `B` were reused.

```
// Parser
WS :    /* empty */
     | WS "TABSSPACES"
     | WS "RETURN";
```

```
// Lexer
TABSSPACES : "[ \t]+"
RETURN : "\n"
```

Although the resulting parse tree records `WS` nodes (which are used for rendering and for ensuring cursor behaviour works as expected), they soon clutter visualisations of parse trees to the point that one can no longer see anything else. In the rest of this thesis, `WS` nodes are thus generally elided from parse tree visualisations.

2.7 History management

The storing and recovery of program changes is only a minor contribution in Wagner's thesis and thus its explanation is brief with many details left to the reader to fill in. This section explains my approach to history management. Section 2.7.1 explains the high level concept Wagner describes in his thesis. Section 2.7.2 explains how nodes can be used to track their own changes which Wagner explains in [88, p. 21]. Section 2.7.3 explains my implementation for saving parse tree changes to the history as well as how to efficiently switch between different versions of the parse tree, which is needed for the implementation of undo and redo.

2.7.1 Background

Most text editors provide undo/redo functionality by keeping track of all the changes the user made since opening the document. The changes are stored as editing operations (e.g. 'insertion of text t at line x, column y') which can be reversed (undo) or replayed (redo) to get to earlier or later stages of the program. For example, if the user inserted text the undo operation would delete that text again; redo would re-insert it.

In his thesis, Wagner proposes a different approach that keeps a history of the user changes by storing the resulting parse tree after every change. These parse trees can then be used to jump back and forth within the history of the program. Undoing a change simply loads the parse tree from before that change. Redo loads the relevant parse tree from a later history. Implemented naively, this would require a large amount of memory to store all versions of the parse trees. Fortunately, when using an incremental parser, we only need to store the difference from one parse tree to another. Thus when storing a parse tree, only nodes that have been edited by the user or were altered by the parser need to be saved. When a node is saved, it simply records any changes to it inside a log on the node itself.

While the general idea is clearly explained in Wagner's thesis, its implementation is only partly shown. The following thus describes one approach of how this feature could be implemented. First, each parse tree change is tagged with a monotonically increasing global version number, which is incremented every time the parse tree is altered by the user, and again after the changes have been re-parsed[3]. Even though for undo/redo it would suffice to only store the version after parsing, we also need to store the intermediate version (after editing but before parsing), which is later needed to recover from parsing errors (see Chapter 3). For this reason executing an 'undo' will go back two versions in the parse tree: one to undo parser changes; and one to undo changes made by the user.

2.7.2 Logging changes

Storing the changes of a node, as described by Wagner, is as simple as logging the node's values each time they are changed. First, each node gets a `version` attribute, which stores the nodes current version, and a `max_version` attribute, which is the highest version number of the node[4]. We then add a new dictionary `log` to all nodes which links *(version, attribute)*-pairs to values, where *version* references a version of the parse tree.

[3] We need not worry about overflowing this value. A quick test revealed that it would take over a billion years of continuous and rapid typing to create enough versions to fill 64 bits.

[4] This attribute is mainly used for performance reasons. Since the version of the parse tree can be bigger than the node's maximum version, this avoids having to iterate over all versions of the node to find the maximum.

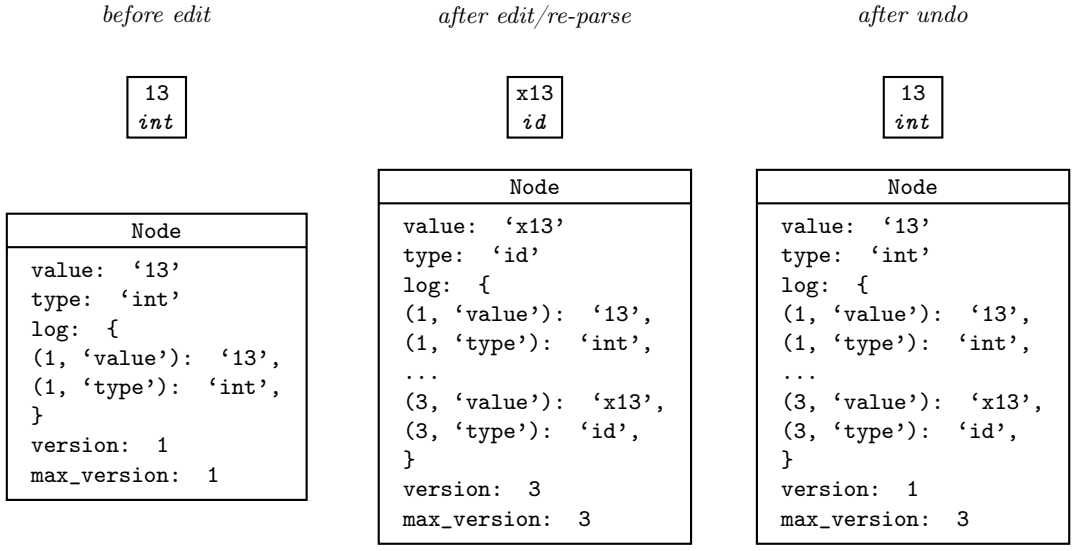


Figure 2.14: A node's log before and after editing, and after an undo. When a node is changed, the changes are stored within the node's dictionary `log`. Upon undo, old values can be retrieved from that log to update the node's attributes. The figure shows a node being edited and afterwards reverted to its initial version. The first row shows the parse tree representation of the node; the second row shows its object representation including some of its attributes.

When a node is changed or re-parsed, its version is updated and changed attributes are saved to the log. Note that changing a node also updates the version of all of its parents up to the root. This means that no node can have a bigger version than the root of the parse tree, or in other words the root of the parser tree is always bigger than or equal to any other node in the tree. There are many attributes that may need to be logged in order to revert the parse tree to a previous state (e.g. besides the value and type, a token also needs to store a pointer to its parent; nonterminal nodes are immutable and thus don't need to store their type, but instead need to keep a log of their children). Reverting a node to an older version can then be achieved by simply reading the old values from the log (see Figure 2.14) and reassigning the node's attributes. Figure 2.15 shows an example of a tree being edited and highlights how the versions of nodes change during editing and re-parsing.

2.7.3 Implementation

Listing 2.9 shows the algorithms for storing and jumping between versions of a parse tree. In order to store the current version of the parse tree we iterate over the entire tree using the function `save_tree`. Each node that is new, was changed, or contains nested changes is saved and its subtree is traversed further. If a subtree doesn't contain any changes, there is no need to traverse it and it can be skipped. Since the attributes that need to be

```
1  def save_tree_D(node: Node, version: int):
2    if node.has_changes() or node.new:
3      for c in node.children:
4        save_tree(c, version)
5      node.save(version)
6
7  def undo_D(node: Node, target: int):
8    if node.version <= target:
9      return
10   for c in node.children:
11     undo(c, target)
12   node.load(target)
13   for c in node.children:
14     undo(c, target)
15
16 def redo_D(node: Node, target: int, _from: int):
17   node.load(target)
18   if node.version > _from:
19     for c in node.children:
20       redo(c, target, _from)
```

Listing 2.9: Functions to save and reload parse trees. The function `save_tree` is called whenever the parse tree changes through user edits or re-parses. It iterates over the current version of the tree and stores all changed and new nodes. Undo and redo work by asking every node in the parse tree (starting with the root) to load the targeted version. During undo, subtrees that already have the target version can be skipped (line 8-9), since loading them would not lead to any changes. We do, however, have to process a node's children both before and after reverting it (line 10-14) for reasons shown in Figure 2.16b. During redo we can skip child nodes, if reverting the parent doesn't change its version, i.e. its version is equal to or smaller than the parse tree version we are reverting from (line 18).

saved and restored depend on the type of the node, each node type implements a method `save` which takes care of storing all relevant attributes to the log (see Listing 2.10).

When the user uses undo to restore the program to an older version, the algorithm iterates over all nodes in the parse tree (starting at the root) and restores all nodes whose version number is bigger than the target version. Subtrees with a smaller version than the target version don't need to be traversed, as they don't contain any changes that need reverting. When a node is reverted we need to traverse its children twice, once before reverting the node itself and a second time after it has been reverted. Since children may have changed between versions it is otherwise possible to miss nodes if the list of children is only traversed once (see Figure 2.16b for an example).

Redo works slightly different to undo. To revert back to a later version we iterate over the tree and attempt to load the target version for each node. If the node doesn't have the target version in its log, it loads the next highest version instead. If the node was reverted then its `version` attribute is now bigger than the version we started with. Only then do we have to traverse its children. This effectively finds all nodes that were created

```
1  class Terminal(Node):
2    def save_Wb(self, version: int):
3      self.log[("symbol", version)] = self.symbol
4      self.log[("parent", version)] = self.parent
5      ...
6
7    def load_Wb(self, version: int):
8      v = min(version, self.max_version)
9      while v not in self.log:
10       v -= 1
11     self.symbol = self.log[("symbol", version)]
12     self.parent = self.log[("parent", version)]
13     ...
14
15   def has_changes_Wb(self):
16     return self.changed or self.nested_changes
```

Listing 2.10: Each node type implements a method `save` which stores all attributes into the node's log. Older attributes can be reloaded via the method `load`. Reverting to version x finds the maximum version y where $y \leq x$ since a) not every node contains every version (line 9-10) b) one can ask a node to load a version that is bigger than its `max_version` (line 8).

or changed during the previous parse. Unlike undo, we cannot skip nodes that already have the target version as that could lead to wrong parse trees (see Figure 2.16c for an example).

2.8 Node reuse

In Section 2.6.2 we have seen how incremental parsing can reuse subtrees if they don't contain any changes, while changed subtrees need to be re-parsed. However, even the re-parsing of subtrees containing changes, can often result in the same subtree as before as Figure 2.17 shows. When logging the history of the parse tree, this can result in the storage of identical data multiple times over, leading to unnecessary memory usage. The problem gets worse, the more frequently we re-parse the tree, for example after every keypress.

We can minimise this problem with *node reuse*, which identifies such subtrees and, instead of generating new nodes, reuses nodes from the previous parse tree. In his thesis, Wagner proposes two solutions to this problem: *bottom-up* and *top-down node reuse*. Both solutions have weaknesses which cause them to miss some opportunities to reuse nodes. However, they complement each other and when run together 'restore virtually every token that the user would consider unchanged' [88, p. 53], according to Wagner. Bottom-up reuse runs alongside the incremental parser and when a reduction occurs reuses nonterminal nodes from the previous version of the parse tree if possible, instead

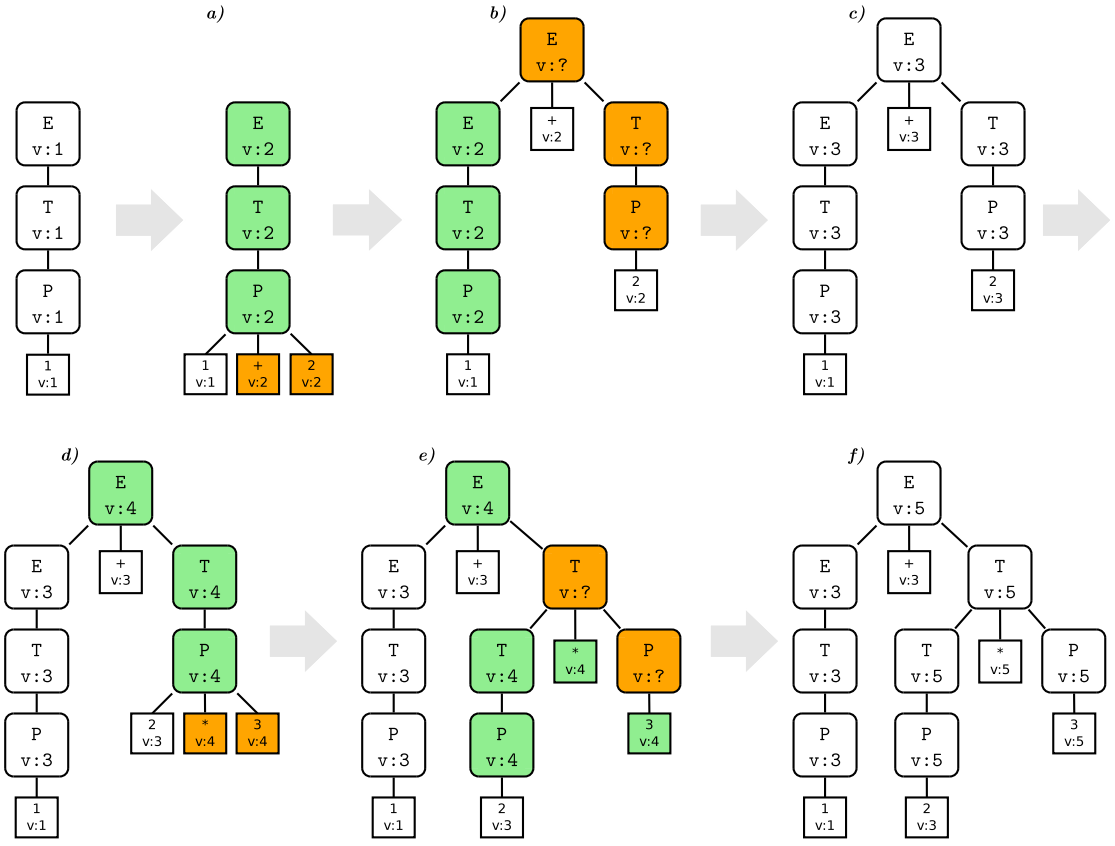


Figure 2.15: Node versioning during editing and parsing operations of a parse tree. Any node that was changed or whose subtree contains changes (green), or was newly created (orange), is saved after each editing and each parsing operation. This updates their version numbers, which are shown inside the nodes in the form of '`v:#`'. **a)** The user inserted some input. The parse tree version was increased by 1, and all changed and new nodes stored their values in their log, tagged with the current parse tree version. **b)** The parse tree was re-parsed which generated new nodes and assigned some terminals to new parents. The changes have not been stored to the history yet, so old nodes remain at the same version, while new nodes do no have a version yet. **c)** The changes made by the parser have been stored within the nodes while the parse tree version was increased to 3. Afterwards, the `changed`, `nested_changes`, and `new` attributes of all nodes are reset. **d)** The user added more input and the changes are stored within the nodes, increasing the parse tree version to 4. **e)** The changes have been re-parsed, creating some new nodes and moving some old nodes around. The changes have not been stored yet. **f)** The parse tree version was increased to 5 and all changes were stored within the nodes.

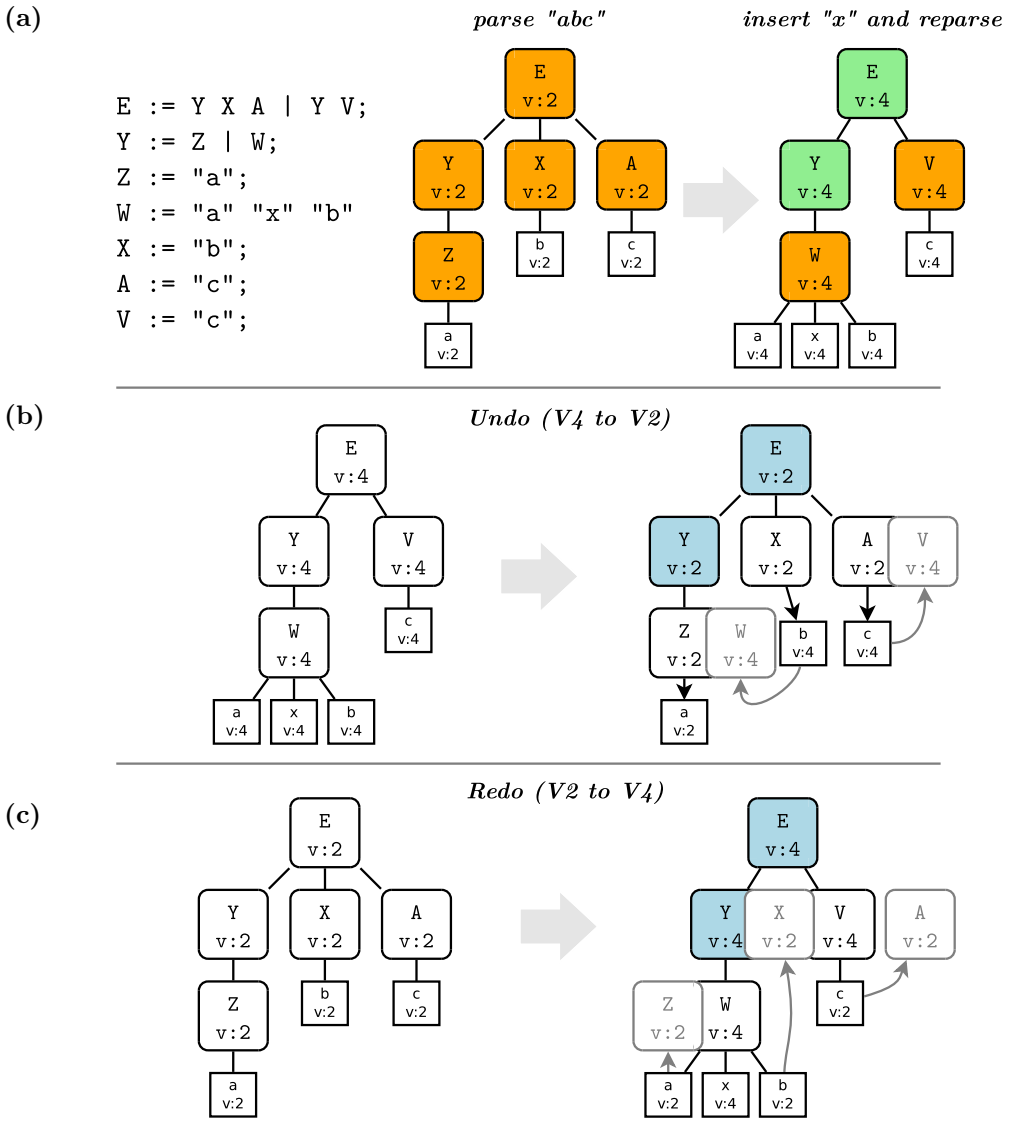


Figure 2.16: An example showing how a naive implementation of reverting subtrees can lead to a wrong parse tree. Parents pointing to their children is visualised via a black arrow. Children pointing to their parent via a grey arrow. A child and parent pointing to each other, is visualised by a black line (no arrow). Nodes that were reverted are coloured in blue. **(a)** Using the grammar on the left for the input `abc` produces the middle parse tree. Inserting `x` between `a` and `b` results in the parse tree on the right. **(b)** This example shows why during undo we need to traverse children both before and after the parent has been reverted. Without it, reverting the parse tree from version 4 to version 2, results in the children `b` and `c` still pointing to their parents from version 4. When we only traverse children after reverting the parent, reverting `E` would set its children to `Y`, `X`, and `A`. The last two already have version 2 and are not traversed further, meaning that `b` and `c` are also not traversed and thus not reverted (note how they remain at version 4). **(c)** This example shows why during redo we need to also traverse nodes that already have the target version. When reverting `E` from version 2 back to 4, this sets its children to `Y` and `V`. Since `Y` is at version 2, it is reverted also, setting its child to `W`. `V` and `W`, however, already are at version 4. If we do not traverse them further, the children `a`, `b`, and `c` would not be reverted, remaining at version 2, and pointing to their previous parents.

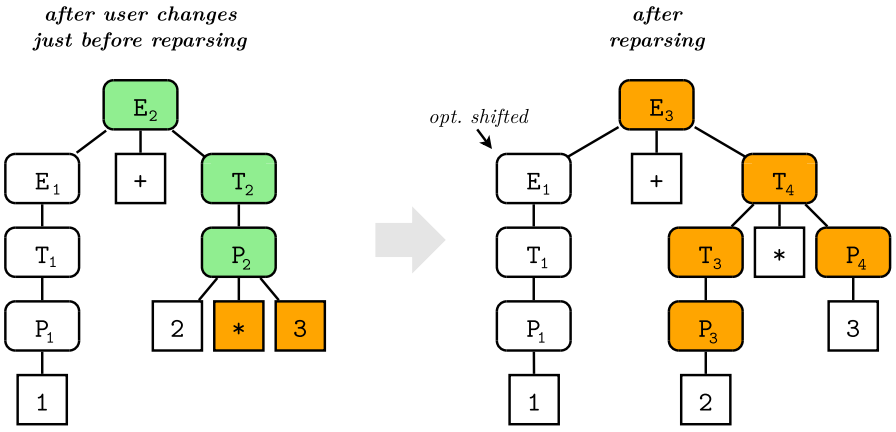


Figure 2.17: An example of re-parsing a tree without node reuse. Since the subtree `T2` had changes, it needed to be re-parsed and couldn't be optimistically shifted. However, re-parsing it results in an identical subtree `T3`. Since all versions of parse trees are stored to its history, this leads to unnecessary memory usage.

of creating a new one. Top-down reuse runs *after* parsing has finished and traverses all newly created nonterminals and attempts to replace them with their counterpart from the previous version of the parse tree, if one exists. It is important to note that while bottom-up reuse prevents nodes from being created in the first place, top-down reuse only replaces them after they have been created, though still before they are saved to the history.

2.8.1 Bottom-up reuse

Bottom-up reusable nodes can be found during parsing whenever a reduction occurs. Upon the reduction, we check if the children being reduced already have a parent in the previous parse tree. If the type of the old parent is the same as the type of the reduction, and all children share the same parent in both the previous and current version of the parse tree, then the old parent can be reused. Wagner calls this *unambiguous node reuse*. He also gives a more relaxed form of bottom-up reuse, called *ambiguous node reuse* which doesn't require all children to share the same parent; instead a single child whose old parent matches the new reduction type is sufficient to reuse that parent. However, this requires remembering which nodes have already been reused so that they are not reused multiple times within a single parse. This can be achieved with a set, which we add each node to that was reused, and which is reset on every re-parse. Though this is not explicitly stated in Wagner's thesis, note that when reused nodes are broken down via `right_breakdown` they need to be removed from the set so that they can be reused in any upcoming reductions when parsing continues.

```
def ambig_reuse_check_W(prod: Symbol, children: List[Node]) -> Node:
  for c in children:
    if not c.new: # not a new node
      old_parent: Node = c.get_parent(previous version)
      if old_parent.symbol == prod and old_parent not in reuse set:
        add old_parent to reuse set
        old_parent.set_children(children)
        return old_parent
  return Node(prod, children)
```

Listing 2.11: To find a reusable nonterminal the algorithm iterates over all children that are being reduced and checks their parent nodes from the previous version of the parse tree. If any of the parents has the same production symbol as the current reduction and has not already been reused, it is returned to the parser. Otherwise a new node needs to be created.

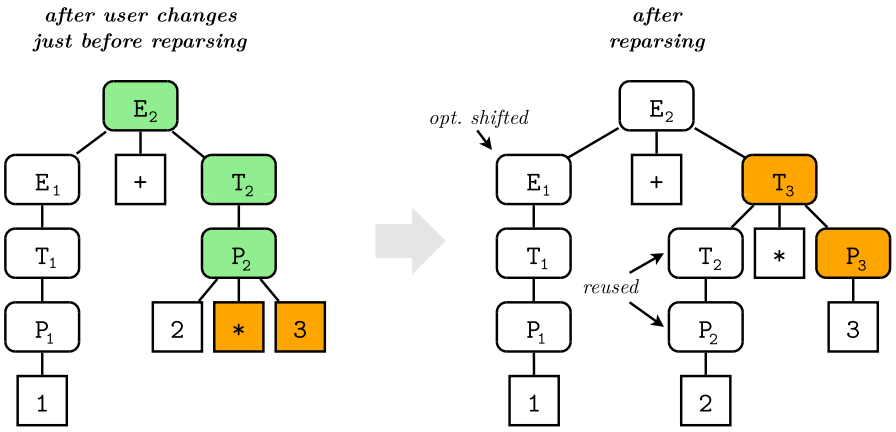


Figure 2.18: re-parsing user changes in a parse tree with bottom-up node reuse enabled. We can see that both the nodes T_2 and P_2 could be reused by the incremental parser and didn't need to be recreated during their re-parse.

Since ambiguous bottom-up reuse is more effective in finding reusable nodes than unambiguous node reuse, only the former is discussed here. Adding bottom-up reuse is straightforward. Within the `reduce` method of the incremental parser, we replace the creation of a new nonterminal node with a call to a function `ambig_reuse_check` that aims to find reusable nodes using ambiguous bottom-up reuse. This function takes the current production and the children that were popped from the stack, and returns a parent node from the previous parse tree with the same production type, if one exists, or returns a new node if not. Reusing a node and assigning new children to it will mark that node as *changed*, which also causes all of its ancestors up to the root to have *nested changes*. The implementation is shown in Listing 2.11 and Figure 2.18 shows the parsing of a program with bottom-up node reuse enabled.

```
1  def top_down_reuse_W():
2    top_down_traversal(root node)
3
4  def top_down_traversal_W(node: Node):
5    if node.changed and not node.new:
6      reuse_isomorphic_structure(node)
7    elif node.nested_changes or node.new:
8      for c in node.children:
9        top_down_traversal(c)
10
11 def reuse_isomorphic_structure_WD(node: Node):
12   for i in range(len(node.children)):
13     current_child = node.children[i]
14     if i >= len(node.get_children(previous version)):
15       top_down_traversal(current_child)
16       continue
17     else:
18       previous_child = node.get_children(previous version)[i]
19     if current_child.new and not previous_child.exists and \
20         current_child.symbol == previous_child.get_symbol(previous version):
21       replace_child(node, i, current_child, previous_child)
22       reuse_isomorphic_structure(previous_child)
23     elif current_child.nested_changes:
24       top_down_traversal(current_child)
```

Listing 2.12: Improved Python version of Wagner's top-down reuse algorithm. Top-down reuse runs after the parse tree has been fully re-parsed. It traverses the tree from top to bottom, following all nodes that contain *nested changes* (lines 7–9). If a node has local changes (shown by the `changed` attribute), this means that it has different children than in the previous version, some of which could be newly created nodes. We then analyse the node's children and compare each with the child at the same index in the previous version of the parse tree (lines 19–20). If their symbols are the same, the new child can be replaced with the old one (line 21). Afterwards, the algorithm continues traversing the parse tree.

2.8.2 Top-down reuse

Even though bottom-up reuse can recover the majority of reusable nodes, some elude the algorithm. As we have seen in Listing 2.11, in order for bottom-up reuse to find a reusable node, it inspects the parents of the children being reduced. This means that empty nonterminals (i.e. nonterminals without children, produced from ϵ-rules) can never be found using bottom-up reuse. However, these nodes can be discovered and replaced using top-down reuse. Top-down reuse runs after parsing is complete. It traverses the parse tree from top to bottom and compares any newly created node with the node that shares the same location in the previous version of the parse tree. If they both have the same type (i.e. both have the same production symbol), then the new node can be replaced with the previous one. Replacing a new node with a node from the previous version of the tree means reassigning children from the new node to the old node and copying over important attributes. A modified version of Wagner's algorithm for top-down reuse is shown in Listing 2.12.

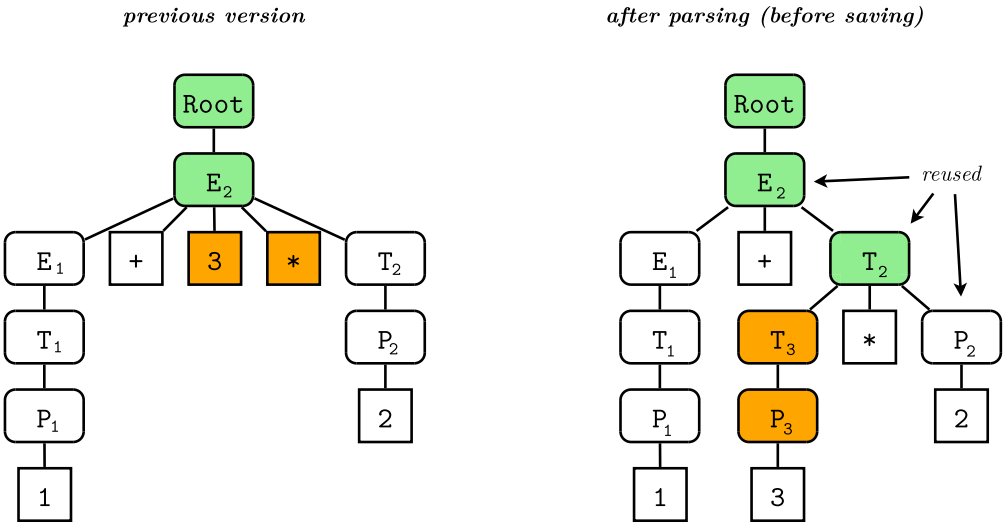


Figure 2.19: This example shows that nodes can have different numbers of children between the previous and the new version of the parse tree. The node `T2` has 3 children in the new parse tree, but only one in the previous. This would result in a runtime error in Wagner's original algorithm when trying to access `T2`'s second child from the previous version. For this reason we need to add an extra check in `reuse_isomorphic_structure` for cases where the previous version of a node has less children that is currently has.

Wagner's original algorithm [88, p. 75] has two problems. The first is that it assumes that the previous and current version of a node always have the same amount of children. Figure 2.19 shows that this is not always the case and using the original algorithm on such nodes can lead to runtime errors. It is unclear if this is a flaw in Wagner's algorithm or a mistake introduced accidentally during the simplification of the code example in his thesis, though the latter seems more likely. The improved implementation shown in Listing 2.12 fixes this issue by adding an additional if-else-block (lines 14–17) to catch cases where the previous version of a node has less children than the current version. In those cases we simply skip the remaining children and continue to traverse the rest of the tree (line 15).

The second problem is that the original algorithm only traverses the children of a node, if that node has nested changes. However, new nodes can also contain subtrees with changes in them, which need to be traversed to find all reusable nodes (see Figure 2.20). Even though it is not explicitly stated in his thesis, I assume that in Wagner's implementation new nodes inherit the `nested_changes` value from their children when they are created. For implementations where this is not the case, the algorithm in Listing 2.12 was changed at line 7 to include new nodes in the traversal as well.

Note that reusing tokens from a previous version is not always possible depending on the frequency in which an editor stores the parse tree's history. Wagner describes the use case for reusing tokens as a user removing a token from the tree and then immediately

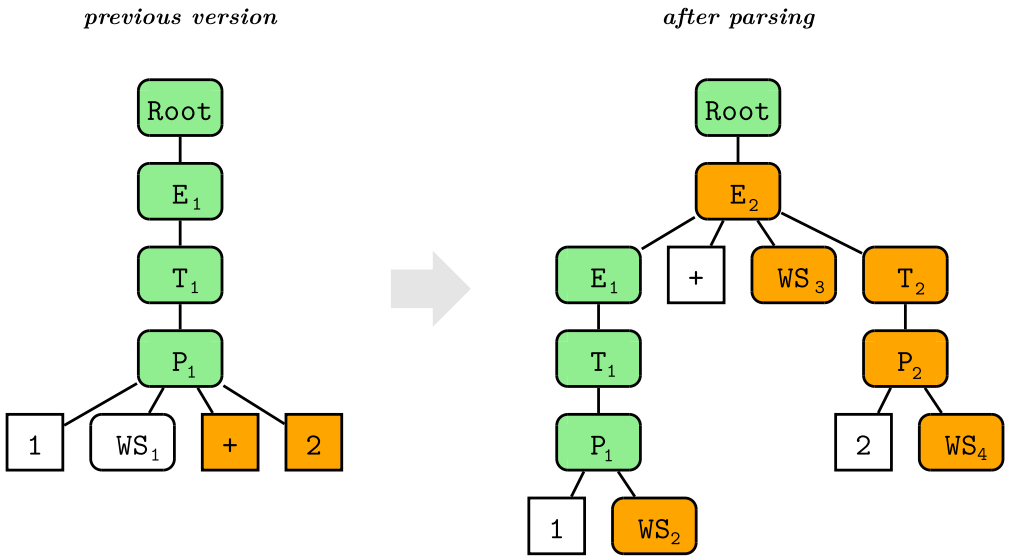


Figure 2.20: This example shows why it is necessary to also scan new nodes during top-down traversal. Let's assume the calculator grammar has been extended with optional whitespace between terminals. After parsing the user edits, we get a new nonterminal node `E2`. Its subtree contains nodes that were changed during parsing, including the newly created node `WS2`. Using top-down reuse, this node can be replaced with its counterpart from the previous version. However, this requires traversing the subtrees below new node `E2`.

re-entering it. In such cases top-down reuse can find the previous token and then replace the new one. However, an editor may choose to store parse trees after each keypress, creating two versions, one after the user edited the parse tree, and a second after it has been re-parsed. Thus, when the user deletes a token, the change is immediately saved, then re-parsed, and then saved again. If the user then re-enters that token, the parse tree is saved yet again prior to re-parsing. By the time top-down reuse is called, we are already 3 versions ahead and the newly inserted token will have already been saved to the history, which means that replacing it with a previous token won't lead to any performance or memory gain.

2.8.3 Evaluation

In order to evaluate the effectivness of node reuse, I ran a small experiment, analysing node usage for some small programs with and without node reuse. The results show that even on small inputs these reuse algorithms are capable of saving up to 83% of nodes from being recreated, depending on input and grammar. For example, typing out the Python program from Listing 2.13a without node reuse leads to a total of 760 nodes being created. Bottom-up reuse reduces this number by 63% to 284; top-down reuse reduces those 284 nodes by another 47% down to 134. Together they save 83% of nodes from being saved to the history. Typing out the Java program from Listing 2.13b without

```
class X:
  def x():
    pass
```

(a) Python program

```
class X {
  public static void main(){
    method();
  }
}
```

(b) Java program

Listing 2.13: The two programs used to benchmark node reuse. Using bottom-up and top-down node reuse saves 83% of nodes from being stored to history for the Python program. For the Java program that number is slightly smaller, saving about 68%. While these examples are small we can expect similar figures for larger programs as well.

node reuse generates a total of 1221 nodes. Bottom-up reuse reduces this number by 39% down to 742; top-down reuse takes off another 47% down to 389 nodes. Together they reduce the number of nodes being saved to history by 68%.

2.9 Performance evaluation

In order to assess the performance of incremental parsing I conducted an experiment using the Java standard library. In this section I outline the methodology (Section 2.9.1) and discuss the results (Section 2.9.2).

2.9.1 Methodology

Measuring incremental parsing performance is difficult as it depends on where the program was edited before it is re-parsed. The challenge thus becomes how to make enough plausible edits to a file so that we can obtain meaningful statistics. My approach was to automatically edit a file in multiple locations and measure the average re-parse time. The experiment was run on the Java 1.5 standard library using *Eco*'s incremental parser implementation and a Java 1.5 grammar. The library consists of 6560 files totalling 222KLoc, ranging from 391 bytes (43LoC) to 200 Kbyte (5664LoC) in size.

First, each file was loaded into *Eco* and an initial parse tree created. The file was then automatically edited by adding '`1+`' to each assignment in the program (i.e. after every '`=`'), forcing the program to be re-parsed incrementally. For each edit I measured the time needed to re-parse the changes and update the parse tree, excluding any processes before and after parsing such as incremental lexing or top-down reuse. I then recorded the average performance of all edits. If an edit lead to a parsing error, no more edits were made to that file and only the results up to that point were recorded, as the performance of further edits would be slowed down by error recovery.

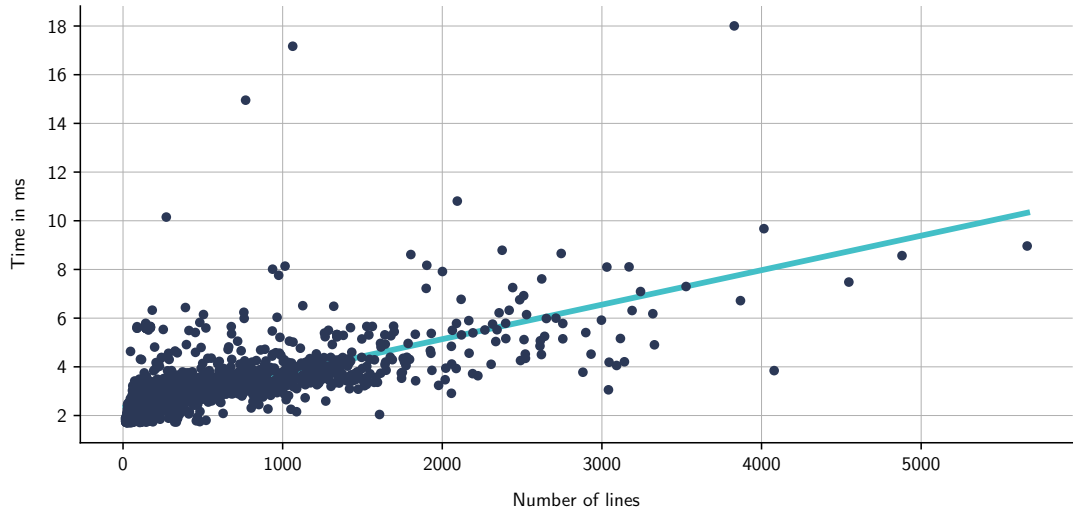


Figure 2.21: Diagram showing the performance of incremental parsing depending on program size. The x axis shows the number of lines for each program measured, the y axis shows the average time spent re-parsing edits within each program. The blue line shows the best fit function for the data.

All experiments were run on an otherwise unloaded Intel Xeon E3-1240 v6 with 32GiB RAM running Debian 9. The repeatable experiment can be found at `https://github.com/softdevteam/eco_benchmark`.

2.9.2 Results

Figure 2.21 shows the results of the experiment. The conclusion is clear: the time taken by the incremental parser correlates with the number of lines in a file. In other words, the time to re-parse parts of a file is partly determined by the overall size of the file. However, the increase is sub-linear: roughly speaking, doubling the size of the file slows the incremental parser down by only 25%. The majority of files in the Java standard library lie within the 1500LoC range where *Eco* is capable of re-parsing edits within 4-6ms, and even files with over 4000LoC stay below 10ms. Even though humans can perceive latency of as little as 2ms [58], these results are well within an acceptable range. Given that *Eco* uses the standard Python interpreter to run, and that implementation efficiency was not a particular goal, it is clear that implementations in faster languages would be able to reduce this figure further. In summary, the results show that incremental parsing can be used to parse programs after every keypress even for larger programs without causing a significant increase in latency that hinders the editing in such environments.

The results include a few outliers of which three seem of particular interest, given that their runtime is relatively high compared to other files with a similar amount of line

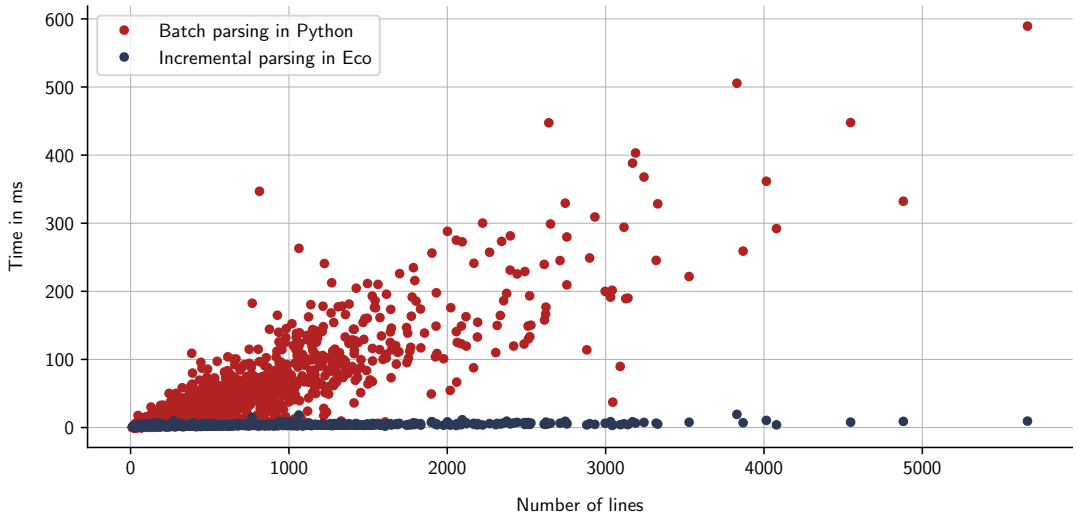


Figure 2.22: Diagram comparing the performance of incremental parsing in Eco and batch parsing in Python. The x axis shows the number of lines for each program measured, the y axis shows the average time spent re-parsing edits within each program.

numbers. Investigating the corresponding files reveals that they trigger a worst case scenario for incremental parsing. Two files (`XMLChar.java` and `Constants.java`) contain blocks with hundreds of statements, while the other (`OMGSystemException.java`) contains a class with a large number of functions. Parsing such programs creates deeply nested parse trees, due to the left recursiveness of lists in LR grammars. For example, in Java the rule for a list of statements has the form '`statements: statements statement | statement;`', and thus each statement increases the parse tree's depth by one. When editing a statement, the parser needs to break down the path from the root down to the edit in order to reach it, and even though the statement subtree itself can be incrementally re-parsed, all reductions back to the root need to be reapplied. This effect is worse the earlier the statement appears, as it is then located deeper in the parse tree. Thus, editing the first of the statements has the worst runtime, which in case of `XMLChar.java` is about 31ms. The runtime decreases with every following statement that is being edited as they are located higher in the parse tree. Other edits within the file have a normal runtime of about 2ms, thus pushing down the average incremental parsing runtime of the program to around 17ms as seen in the figure.

2.9.3 Comparison to batch parsing

This section compares the performance of incremental parsing in *Eco* to traditional batch parsing in Python. The aim of this comparison is to make the two parsers as directly comparable as is practical, though there are some inevitable differences. The methodology for the incremental parser is the same as in Section 2.9.1. However, since a batch parser

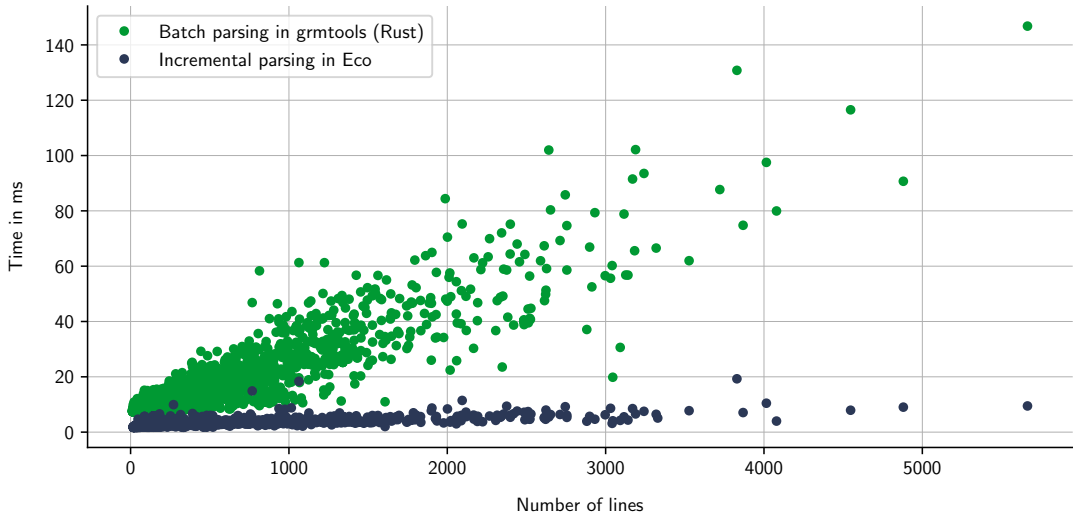


Figure 2.23: Diagram comparing the performance of incremental parsing in Eco and batch parsing in grmtools (Rust). The x axis shows the number of lines for each program measured, the y axis shows the average time spent re-parsing edits within each program.

always parses the entire program from scratch, it isn't necessary to edit the program between parses, so instead the same program is simply parsed multiple times, recording the average. The measurements do not include time spent reading the file from disk or lexing its contents, which benefits the batch parser which, unlike *Eco*, would still have to do these steps in a normal setting. The batch parser also doesn't generate a parse tree, giving it even more of an advantage. Figure 2.22 shows the results of the two parsers when used on the Java Standard Library. Despite a methodology in the batch parsers favour, we can still clearly see that batch parsing scales badly with increasing program sizes, compared to incremental parsing. The batch parser is orders of magnitude slower than the incremental parser, reaching parsing times of over 100ms for files with only a few hundred lines of code. Bigger files (e.g. 1000+ LoC), increase this further to 200ms and more, making it difficult to parse these programs after every keypress.

Some readers may suspect that comparing a slow Python batch parser against a slow Python incremental parser tells us little of interest. What happens if we compare our Python incremental parser to a fast batch parser written in a compiled language? Figure 2.23 shows a comparison of incremental parsing in Eco and batch parsing in Rust (generated using grmtools [75]), used on the Java Standard Library. Similar to the traditional batch parser in Python, instead of editing the program, the grmtools batch parser is simply run multiple times on the same input and the average recorded. While the grmtools parser also doesn't generate a parse tree, the timings do include lexing, bringing it a little bit closer to the actual runtime of a batch parser when used within an

editor[5]. The results show that, though a factor of 10 times faster than the Python batch parser, the grmtools parser is still an order of magnitude slower than incremental parsing in Eco. In the worst cases we are still getting parse times of over 100ms, which would be unacceptable for an editor that parses on every keypress. We can therefore conclude that incremental parsing is still a useful and neccessary technology even with the speed of modern processors.

2.10 Related work

Many existing projects today employ some form of incremental parsing. Both Roslyn [55] and Atom [14], for instance, use Wagner's incremental parsing algorithms, with minor differences. Roslyn is a compiler platform, that provides tools, like compilers and code analysis, for programming editors such a Visual Studio. Since it communicates via an API, editing operations by the user are not directly applied to the parse tree. Instead, users edit normal text, which is then first synced with the parse tree maintained by Roslyn and then incrementally re-parsed. Atom is a popular programming editor written in JavaScript. The main difference is its incremental lexing approach, which instead of using lookback counts, uses checkpoints from which lexing restarts after the user has edited the program. Other projects, like Eclipse [46] or Papa Carlo [47], use ad-hoc, language-specific variants of incremental parsing, which only incrementally re-parse small portions or fragments of a program (e.g. blocks). Such fragments can not be easily generalised and need to be defined manually for each language. They also do not have the granularity of Wagner's approach, and thus often re-parse more than is necessary.

Comparable incremental parsing algorithms exists for other grammar types. Shilling [72], Li [52], and Yang [90] have developed incremental parsing techniques for LL parsers, though, according to Wagner, their algorithms suffer from problems such as being restricted to a single editing site, generating incorrect parse-errors on ϵ-rules, or having non-optimal performance [88, p. 57]. Dubroy et al. show how a packrat parser can be extended with incremental parsing, by using the memoization table to reuse results from previous parses [22]. However, their approach is more conservative and only reuses subtrees that weren't changed; any changed node needs to be recreated, even if it ends up being the same.

Not to be confused with incremental parsers are online parsers, which can parse a stream of partially known input as it becomes available, e.g. Yi [9] and differentiating parsers [43]. For example, Yi only parses the input that is currently visible on the screen. This has the

[5]For the sake of completeness, Appendix A shows the comparison of *Eco*'s incremental parser and the grmtools batch parser *without* lexing.

advantage, that when opening a large file, it can be parsed very quickly. The remainder of the file is then incrementally parsed as the user scrolls down, utilising parsing checkpoints and caching previous results. The downside of this is that if the users jumps back and forth between the beginning and end of the file (while editing the beginning) the whole file needs to be re-parsed, as previously cached results become invalid.

Chapter 3

Error recovery

Error recovery is an important feature for any editor that wishes to show more than only the first parsing error of a program to the user. Unfortunately, traditional error recovery algorithms have many shortcomings, e.g. they are language dependent or their results can often be more misleading than helpful in understanding the cause of an error. Disregarding this, to the best of my knowledge no traditional error recovery approach has been adapted to work within an incremental parser, and due to the complexity of the subject, doing so was out of the scope of this thesis. I thus use Wagner's history-based error recovery algorithms for incremental parsers. This chapter discusses some of its details that are incompletely or ambiguously explained and also describes some problems Wagner's algorithms have with certain grammars and provides solution for those problems. Although all of Wagner's error recovery algorithms are summarised, their explanations are limited to only the most crucial details that are necessary to understand the problems described here. If the reader wishes to know more about the finer details of the algorithms, they may want to refer to Wagner's description of his techniques [88, p. 91]. Some of the solutions discussed here require slight modifications to the incremental parsing algorithm. Appendix C summarises the full incremental parsing algorithm, with these changes applied.

3.1 Introduction

In a batch environment there are several ways to recover from an error. For example, in YACC, language creators can leave hints within the grammar which tell the parser what to do in case an error occurs. By adding the symbol *error* to a grammar rule, the author can tell the parser to skip all input until a certain terminal symbol is found, and then reduce the rule as if the correct input had been parsed. The parser can then continue

parsing as usual. The downside to this method is that it is language dependent, needs manual alteration of the grammar, and may require multiple attempts to to get satisfying results. An alternative, language independent, and automatic method for error recovery is to make the parser guess terminal symbols to remove from or insert into the input so that the parser can continue parsing [17]. The disadvantage of this approach is that there are multiple ways in which an error can be repaired and it isn't always obvious which repairs should be preferred, leading to cascading errors which can often be misleading [20]. Often the location of an error and the change that caused it are far away from one another which makes it difficult to generate sensible error messages. Some studies thus find that error messages often fail to aid programmers, especially novice ones, to understand the cause of an error [59, 76].

In his thesis, Wagner proposes a complementary approach to traditional error recovery that takes advantage of parse trees and their history. The history of the parse tree includes every change the user made since the last time the program was parsed. Using this history it's possible – with a few exceptions like loading invalid programs or pasting large chunks of code, where we need to fallback to traditional error recovery – to precisely pinpoint the occurrence of an error to the exact change the user made that lead to it. If a program was valid and becomes invalid after the user edited the program, e.g. via the insertion or deletion of code, we know that edit must have caused the error.

In order to recover from an error, Wagner's solution is as simple as pretending that the change that caused the error never happened. This can be achieved by skipping the subtree containing the edit during parsing, i.e. pushing the subtree from the previous version of the tree onto the stack without looking at any changes contained within it. Wagner calls this *isolating* the error. This is similar to manually removing the change that caused the error, parsing the program, and then inserting the change back into the program. Once a subtree has been isolated it does not need to be inspected again in subsequent parses unless it contains new changes or its surrounding context[1] has changed. When isolating a subtree, we refer to the root of that subtree as its *isolation node* and mark it as such.

Isolating errors allows us to continue parsing any other changes the user made that follow the isolation and integrate them into the parse tree. The downside is that if the isolation area is very large – at worst the entire parse tree can be isolated – a large part of the program is not analysed. This prevents user changes within the isolated subtree from being parsed and integrated into the parse tree or, if the changes are invalid, hide those errors from being shown. User changes within isolation trees occur in two locations:

[1] In short, *surrounding context* means a node's lookahead that is used to determine the reduction rule of the node. See Section 3.5.1 for a more detailed explanation.

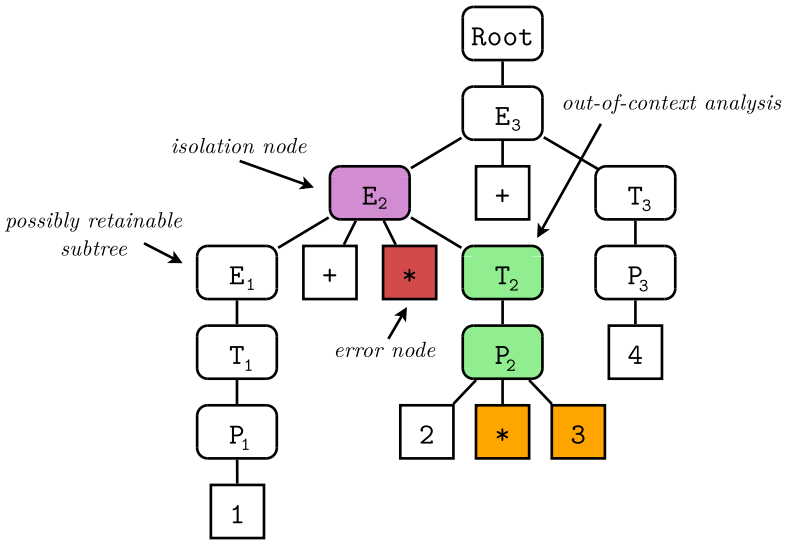


Figure 3.1: A parse tree where subtree `E2` has been isolated to recover from a parsing error at node '`*`'. If subtree `E1` had changes, they would have been re-parsed by the parser and could potentially be retained. The subtree at `T2` contains some valid user changes that haven't been re-parsed yet due to the error. However, we can use *out-of-context analysis* to independently parse and integrate those changes into the parse tree. After the isolation of `E2`, the parser continues to parse the remainder of the parse tree as normal.

before and after the error node. Changes before the error node will, by definition, have been parsed normally. However, isolating the subtree they are contained in discards those changes again, since their proximity to the error suggest they are likely to be wrong. Changes within the isolated subtree, that come after the error, will not have been parsed. In his thesis, Wagner thus also presents solutions to retain structural changes before and analyse changes after the error. The former is done by traversing all changed subtrees before the error, *retaining* valid changes while discarding the rest. The latter is achieved using *out-of-context analysis*, which runs an independent analysis (using a separate parser) on all affected subtrees and merges the result back into the main tree. Figure 3.1 explains the terminology with the help of a parse tree.

The remainder of this chapter follows the same structure as Wagner's thesis and explains his error recovery in more detail. It also corrects a few typos, discusses problems with empty nonterminals in several parts of his algorithms and how they can be solved, and improves upon node reuse and tree traversal during error recovery. We will use Wagner's terminology to name specific versions of the parse tree. The term *previous version* means the version of the parse tree after the user has made changes, but before those changes have been re-parsed. The term *current version* describes the parse tree on the parsing stack as it is being constructed. This version may be incomplete, which means it consists partially of old and new subtrees, some of which may not have been attached to a parent yet. The *reference version* describes a parse tree, preceding the previous version, that is syntactically correct, i.e. has no errors, and has no pending user changes.

```
1  def find_iso_tree_W() -> (Node, int):
2    node = stack[-1]
3    while node is not root:
4      node = node.parent
5      offset: int = get_offset(node)
6      i: int = get_cut(offset)
7      if i >= 0 and can_shift(node, i):
8        return node, i
9    return root, 0

10 def get_cut_W(offset: int) -> int:
11   sl = 0
12   for i in len(stack):
13     if sl == offset:
14       return i
15     sl += stack[i].text_length()
16   return -1
```

Listing 3.1: Simplified Python version of Wagner's algorithm for finding isolation nodes. The algorithm starts with the top element of the stack (line 2) and traverses its parents (line 3–4). For each parent it tries to find an index on the stack where that parent can be shifted (line 5–7). If such an index is found the parent is returned as an isolation candidate (line 8). To find this index, Wagner uses the method `get_cut`, which iterates over all elements on the stack (from bottom to top) summing up their text-lengths until they match the textual offset of the node being checked. In the worst case scenario that no candidate could be found, the entire tree is isolated (line 9). The method `can_shift` is used to test if a candidate can be shifted at the stack position returned by `get_cut`. When a candidate has been found, the stack is cut back to that position, i.e. all elements after index i are removed. The method `get_offset` calculates a node's textual offset in the parse tree.

Section 3.2 explains how isolation nodes are found and discusses a problem with empty nonterminals that leads to larger than necessary isolations. Section 3.3 explains how already re-parsed changes within an isolation can be retained. It also fixes some typos in Wagner's code examples, adds a minor optimisation that improves tree traversal, and discusses functions from Wagner's algorithms that are insufficiently explained. Two problems are also discussed: one involving the retaining of empty nonterminals which can break error recovery; another regarding the interaction between retained subtrees and history management. Section 3.4 explains Wagner's out-of-context analysis which attempts to parse and integrate unprocessed changes within the isolation area. The section provides an alternative way of setting up out-of-context analysis that, unlike Wagner, doesn't require transforming the grammar. Finally, Section 3.5 discusses post-isolation, e.g. how errors, once they are found, are marked and presented to the user. The section also shows a problem with isolated subtrees in subsequent parses as well as an optimisation that improves node reuse during error recovery.

3.2 Finding isolation nodes

Wagner explains how isolation regions can be computed in [88, p. 96]. Note that the following explanation only gives an overview and leaves out a few details of Wagner's algorithm. Please refer to his thesis for the complete algorithm.

Listing 3.1 shows a highly simplified algorithm for finding isolation nodes. The basic idea for isolating an error is to find a minimal subtree that spans the changes that lead to

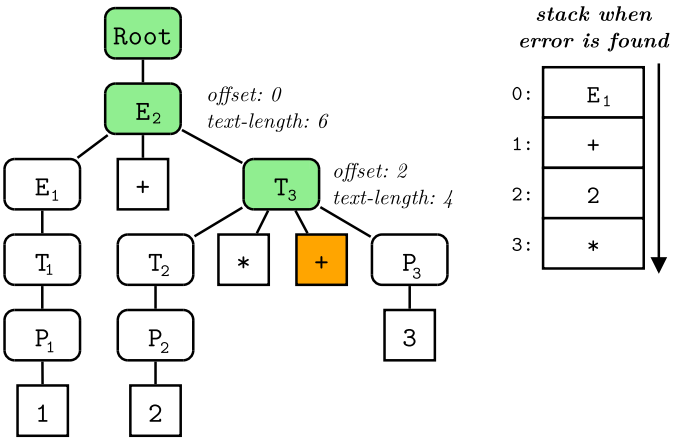


Figure 3.2: An example using the algorithm from Listing 3.1 to find the isolation node for an error. It shows a parse tree (left) where a new token '+' was inserted, and the parse stack (right) at the point an error was found during the re-parsing of the tree. To find an isolation node we start with the node on top of the stack, here '*', and traverse its parents, until we find a valid isolation node. The first candidate we try is T_3, whose offset is 2. Now we need to find a stack position that matches that offset. We do this by iterating over the stack and adding together the text-lengths of each element until it is equal to the candidate's offset. We find such a position at stack index 1, since the combined text-lengths of E_1 and '+' is 2. We then temporarily cut back the stack to that position (in this example this removes '*' and '2' from the stack) and test if the candidate can be shifted. If no such stack position can be found or the candidate couldn't be shifted, we revert the stack and increase the isolation area by continuing to traverse the candidate's parents. In this case, however, T_3 can be shifted and is thus returned as a valid isolation node.

the error, and ignore it so that parsing can continue. The root of this subtree is called the isolation node. To find such a subtree, we first traverse all ancestors of the node immediately before the error. For each ancestor we calculate its textual offset in the parse tree, which is the combined text-length[2] of all tokens in the parse tree, up to (but excluding) the ancestor. We then try to find an index on the current parse stack, where the ancestor can be shifted. This is an index i, where the combined text-length of all nodes on the stack up to (and including) i, is equal to the ancestor's textual offset. Once such a position is found we cut back the stack, i.e. remove all elements from the stack past index i, and check if the ancestor can be shifted there. If no such position could be found or the ancestor cannot be shifted, we revert the stack back to its original configuration and search for a larger isolation area by searching further up the tree. Figure 3.2 shows an example of the algorithm finding an isolation node for an error.

3.2.1 Dealing with empty nonterminals on the stack

Wagner's algorithm for finding isolation nodes does not account for empty nonterminals (i.e. nonterminals without children) on the stack. If the isolation candidate is preceded by

[2]The text-length of a token is the character length of its value.

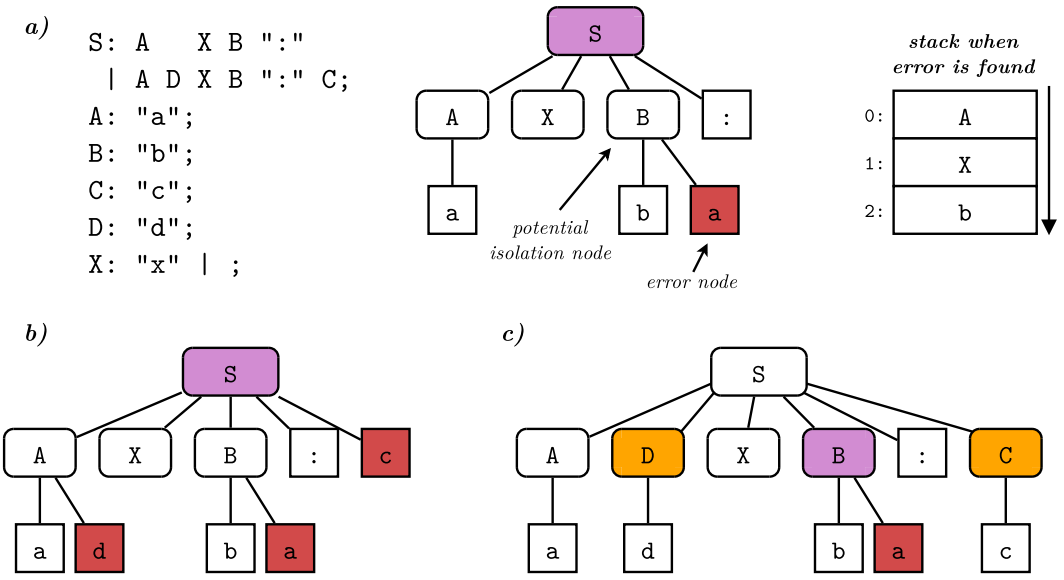


Figure 3.3: An example showing why we should consider empty nonterminals during the finding of an isolation node. **a)** A grammar (left), its parse tree with an isolated error (middle), and the parse stack at the point when the error was found (right). The parse tree shows that even though subtree `B` is a valid isolation, a larger isolation `S` was chosen. While the algorithm considered `B` as an isolation node, finding an index on the stack returned index 0, since `B`'s textual offset is 1 which matches the text-length at index 0 (i.e. subtree `A`). However, `B` cannot be shifted at that position, since the grammar requires `X` to be pushed first. But index 1 was never considered because the text-lengths already matched at index 0. **b)** Choosing a larger isolation area can keep valid changes from being integrated into the parse tree and instead being marked as errors. After isolating the error from (a), the user inserted `d` and `c`. The parser successfully parsed `d`, but the changes cannot be retained, since new subtree `D` doesn't exist in the previous version of the tree. This causes the out-of-context analysis of `c` to fail, since it requires `d` to be parsed successfully. **c)** When choosing a smaller isolation subtree `B`, the user changes can be integrated normally by the parser without the need to retain or out-of-context analyse them.

an empty nonterminal, this can lead to minimum isolation nodes being falsely rejected, and a larger isolation area chosen instead (see Figure 3.3a). Typically, choosing a larger isolation is not too problematic, since the retain algorithm and out-of-context analysis take care of integrating changes within isolations, and 'increased time spent searching for a tighter isolation region may provide little practical benefit' [88, p. 102]; though it may affect the performance of error recovery, since more subtrees need to be retained or analysed via out-of-context analysis. In some cases, however, it can mean that subtrees are not retained or out-of-context analysed at all (see Figure 3.3b). The main reason for these problems is the function `get_cut` [88, p. 97], which stops searching for an eligible stack position as soon as it finds one where the candidate's offset matches the text-lengths of the nodes up to that position. If the isolation candidate cannot be shifted at that position, no other position is tried, and instead the algorithm increases the isolation area by searching the candidate's ancestors. However, if the next stack position has an empty

```
1  def find_iso_tree_WD() -> (Node, int):
2    node = stack[-1]
3    while node is not root:
4      node = node.parent
5      offset: int = get_offset(node)
6      sl = 0
7      for i in len(stack):
8        if sl == offset and can_shift(node, i):
9          return node, i
10       elif sl > offset:
11         break
12       sl += stack[i].text_length()
13   return root, 0
```

Listing 3.2: An improved algorithm for finding isolation nodes, that considers empty nonterminals on the stack when testing if an isolation candidate can be shifted. The function `get_cut` has been integrated into the main function (line 7–12). This allows us to keep searching for a stack position, even if the candidate couldn't be shifted at the last one. However, we stop searching as soon as the combined text-lengths on the stack exceed the candidate's offset (lines 10–11). This makes sure that the index is only increased, if the following elements are empty nonterminals.

nonterminal, that position would still match the candidate's offset and could be a valid position to shift the candidate, but it is never tried.

Fortunately, this problem can easily be fixed. If the index returned by `get_cut` doesn't allow the candidate to be shifted, we simply increase the index. The extra time spent on this is negligible as we only consider stack positions with empty nonterminals, and stop as soon as the combined text-lengths are bigger than the candidate's offset. Listing 3.2 shows an improved algorithm of the one from Listing 3.1. Using this algorithm fixes the problem from Figure 3.3a by choosing a smaller isolation subtree, which allows the other user changes to be retained and analysed as Figure 3.3c shows.

3.3 Retaining subtrees

When isolating a subtree, structural changes made by the parser within that tree must be reverted, because we cannot generally assume that those changes are correct due to their close proximity to the error. However, some subtrees within the isolation area may later be recreated verbatim once the error is fixed. Fortunately, we are often able to prevent this needless repetition of work, by retaining subtrees if they meet certain requirements.

Wagner explains this process in [88, p. 100]. The conditions for retaining a subtree within an isolation can be summarised as follows: the subtree exists both in the previous and current version of the parse tree; and its textual offset and text-length haven't changed between the two versions. Naturally, subtrees that have no changes can always be retained (which means no time is spent traversing their children). Figure 3.4 shows an example of

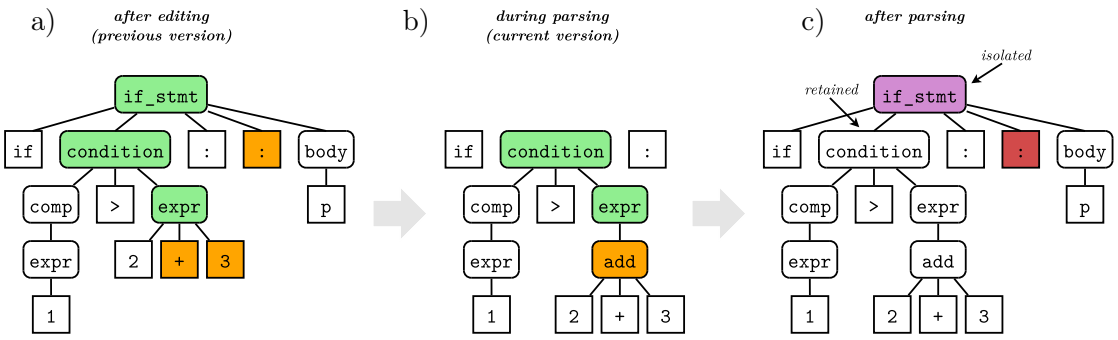


Figure 3.4: An example of an isolated subtree where some changes can be retained instead of being discarded. The parse tree shown is a simplified and elided version of a Python program showing an if-statement with a condition `1>2` and a body `p`. **a)** The parse tree has been edited by the user by adding a second colon and inserting `+3` to the RHS of the condition. **b)** The subtrees that the parser has so far re-parsed and pushed onto the stack, when the error occurred. These are the only candidates that can potentially be retained. We can see that the RHS of `condition` has been re-parsed and a new nonterminal `add` was created. **c)** The parse tree after the error has been isolated. The changes in the subtree `condition` (i.e. the new node `add`) were retained instead of discarded. The subtree `condition` met the retain requirements since its textual offset and text-length are the same as in the previous version.

a subtree being retained during the isolation of an error. The finding and retaining of subtrees within an isolation happens during a refinement process, which also includes out-of-context analysis. Wagner's algorithm is shown in Listing 3.3.

3.3.1 Small optimisation for `pass1`

In Wagner's original algorithm, the function `pass1` attempts to mark any subtree that precedes the error and fulfills certain criteria as retainable, by recursively traversing the isolated subtree. The traversal includes subtrees that span the error, since they may contain subtrees that come before the error that can potentially be retained; but it also includes subtrees that come after the error. The latter, however, can never be retained and thus there is no need to traverse them. Fortunately, we can add a simple optimisation to the algorithm, that stops traversing the parse tree as soon as the current subtree's offset exceeds the offset of the error. The optimisation is shown in Wagner's modified algorithm shown in Listing 3.3 in lines 11–14.

3.3.2 Typo in `pass2`

Wagner's `pass2` function contains a small typo. To traverse the tree, it recursively calls `pass2`. However, instead of passing the child as an argument, the algorithm uses `node` instead. This can lead to infinite loops if the node spans the error offset. Fortunately, the

```
1  def refine_W(isonode: Node, error_offset: int):
2      offset = get_offset(isonode)
3      pass1(isonode, offset, error_offset)
4      isonode.discard()
5      pass2(isonode, offset, error_offset)
6
7  def pass1_WD(node: Node, offset: int, error_offset: int):
8    for child in node.get_children(prev):
9      if offset + child.text_length(curr) <= error_offset:
10       find_retainable(child)
11     elif offset < error_offset:
12       pass1(child, offset)
13     else:
14       break
15     offset += child.text_length(curr)
16
17 def find_retainable_W(node: Node):
18   if node.exists:
19     if not node.nested_changes() or same_text_pos(node):
20       add node to retainable
21       return
22   for child in node.get_children(prev):
23     find_retainable(node)
24
25 def pass2_WD(node: Node, offset: int, error_offset: int):
26   for child in node.get_children(curr):
27     if offset > error_offset:
28       attempt_out_of_context_analysis()
29     elif offset + child.text_length(curr) <= error_offset:
30       retain_or_discard(child, node)
31     else:
32       child.discard()
33       pass2(child, offset)
34     offset += child.text_length(curr)
35
36 def retain_or_discard_W(node: Node, parent: Node):
37   if node in retainable:
38     node.set_parent(parent)
39     remove node from retainable
40     return
41   discard_and_mark_errors(node)
42   for c in node.get_children(curr):
43     retain_or_discard(c, node)
```

Listing 3.3: Wagner's algorithm for refining an isolated subtree. The algorithm has been converted to Python and includes a modification and an optimisation which are described in Sections 3.3.1 and 3.3.2. After an isolation node has been found, `refine` is called with the isolation node as the argument. The algorithm is split into two phases. The first phase (lines 7–15) traverses the previous version (`prev`) of the parse tree and marks eligible subtrees as retainable. Afterwards, the isolation node's changes are discarded (line 4), followed by the second phase (lines 25–41). This phase traverses the current version (`curr`) of the parse tree and runs out-of-context analysis on changed subtrees after the error (lines 28–29), retains changed subtrees before the error if they have been marked as retainable (lines 29–30), and discards changed nodes that span the error (lines 32–33).

```
def same_text_pos_Wb(node: Node) -> bool:
  if text_length(node, prev) == text_length(node, curr) \
        and offset(node, prev) == offset(node, curr):
    return True
  return False

def text_length_D(node: Node, version: int) -> int:
  if node is terminal:
    return len(node.value(version))
  l = 0
  for c in node.get_children(version):
    l += text_length(c, version)
  return l

def get_offset_D(node: Node, version: int) -> int:
  offset = 0
  while node is not root:
    left: Node = node.left_sibling(version)
    if left:
      node = left
      offset += text_length(node, version)
    else:
      node = node.get_parent(version)
  return offset
```

Listing 3.4: Naive implementations to calculate the text-length and offset of a node. The text-length can be computed by iterating over the node's entire subtree and adding together the text-lengths of all contained tokens. A node's offset can be computed, by traversing the parse tree backwards, starting at the node, and adding together the text-lengths of all its left siblings and the left siblings of its ancestors until we reach the root.

fix is simple: the algorithm should pass `child` instead of `node`. The algorithm shown in Listing 3.3 has been changed accordingly.

3.3.3 Efficiently calculating `text_length` and `offset`

One of the conditions to retain a subtree is that its textual offset and text-length are the same between the previous and current version of the parse tree. In Wagner's algorithm this is tested using the function `same_text_pos`, which 'determines whether a subtree's yield occupies the same character offset range as in the previous version of the program.' [88, p. 101]. Its implementation is trivial, however it depends on the offset and text-length of a node, for which Wagner does not provide an implementation. Unfortunately, a naive implementation for those values performs poorly as it requires the repeated traversal of the parse tree even if results are cached (see Listing 3.4). The following explains how we can efficiently calculate offset and text-length values by piggybacking on traversals already carried out by history logging and incremental parsing as well as the retain algorithm.

Both offset and text-length need to be calculated in two versions: the previous version (just after the user made edits) and the current version (the partial parse tree during the re-parsing of the user edits). We can incrementally calculate the current version's text-length of a node during parsing. When a node is created (or reused) during a reduction, all of its children will have already been visited by the parser. We can use this to compute the text-lengths from the bottom up. The text-length of a token is simply the character length of its value. The text-length of a nonterminal is the sum of the text-length of all tokens contained in its subtree. It can be calculated during its reduction, e.g. if one or more tokens are reduced to a nonterminal node, we just add together the text-lengths of all tokens and store the result as the text-length of the nonterminal node. If that node is then further reduced, we can use its stored text-length to compute the text-length of its new parent, and so on.

Note, that these values are only correct for the current version, and editing the parse tree requires them to be recalculated. For example, removing a token from the parse tree changes the text-lengths of all its ancestors. Luckily, prior to a re-parse, user changes are always logged to the parse tree's history as the previous version. Storing those changes requires all changed subtrees to be traversed. This traversal also runs from the bottom up, so we can use it to incrementally calculate each changed node's text-length in a similar fashion to its calculation during a reduction.

Keeping the offset of a node up to date is slightly more difficult. The main reason for this is that an edit at the beginning of the program would change the offsets of every single node in the entire parse tree. Fortunately, the only time this value is needed during refinement is to decide if a node is retainable. This means that it is sufficient to calculate this value during the traversal of the retainable subtrees and only for those nodes that can be retained, i.e. nodes that appear before the error. In Wagner's algorithm we already compute the current version's offset of each of these subtrees during the `pass1` traversal, by adding together the text-lengths of each node we encounter. Similarly, we can calculate the previous version's offset by summarising the previous version's text-length of each node. We then simply pass these values over to the `same_text_pos` function. Listing 3.5 shows the updated functions of the refinement algorithm.

3.3.4 Implementing `node.exists()`

Both the retain and top-down reuse algorithms need to know whether or not a node exists in the current version of the parse tree. In Wagner's implementation nodes have a method `exists(`*`version`*`)`, which returns `false` if that node previously has been marked as deleted using the `process_deletion` function [88, p. 19], which is referenced in the

```
1  def refine_WD(isonode: Node, error_offset: int):
2      offset = stack_offset()
3      pass1(isonode, offset, offset, error_offset)
4      isonode.discard()
5      pass2(isonode, offset, error_offset)
6
7  def pass1_WD(node: Node, offset: int, poffset: int, error_offset: int):
8    for child in node.get_children(prev):
9      if offset + child.text_length(curr) <= error_offset:
10       find_retainable(child, offset, poffset)
11     elif offset < error_offset:
12       pass1(child, offset, poffset)
13     else:
14       break
15     offset += child.text_length(curr)
16     poffset += child.text_length(prev)
17
18 def find_retainable_WD(node: Node, offset: int, poffset: int):
19   if node.exists:
20     if not node.nested_changes() or same_text_pos(node, offset, poffset):
21       add node to retainable
22       return
23   for child in node.get_children(prev):
24     find_retainable_subtrees(node)
25     offset += child.text_length(curr)
26     poffset += child.text_length(prev)
27
28 def same_text_pos_WD(node: Node, offset: int, poffset: int) -> bool:
29   if node.text_length(prev) == node.text_length(curr) and offset == poffset:
30     return True
31   return False
32
33 def stack_offset_WD() -> int:
34   l = 0
35   for n in stack:
36     l += n.text_length(curr)
37   return l
38
39 # called during reduce/save_tree
40 def calc_text_length_D(node: Node, version: int):
41   if node is terminal:
42     l = len(node.value(version))
43   else:
44     l = 0
45     for c in node.get_children(version):
46       l += c.text_length(version)
47   node.set_text_length(l, version)
```

Listing 3.5: Updated functions for finding retainable subtrees, which calculate offsets for each node on the fly, using text-lengths pre-calculated during reductions. Since `pass1` already calculates the current version's offset (line 15), we only have to add the computation of the previous version's offset (line 16); its inital value passed by `refine` is the same as `offset`. Since the function `find_retainable` also traverses the tree, we need to continue computing both the current and previous version's offset (line 25, 26). These values are then simply passed over to `same_text_pos`, where they are used to determine if a node is retainable. The initial offset of the isolation node (second argument of `pass1`) can now also be calculated using `stack_offset` instead of `get_offset`, which simply sums up the text-lengths of the nodes on the stack (lines 33–37).

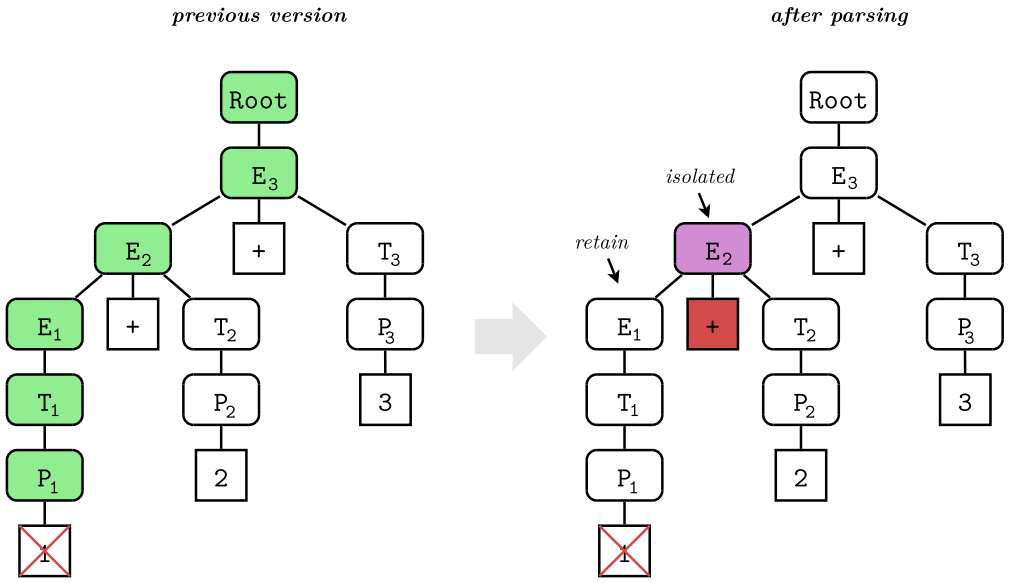


Figure 3.5: This example shows a subtree being falsely retained even though it didn't exist in the current version. The parse tree on the left shows the node `1` being deleted by the user. During re-parsing, an error occurs immediately when trying to parse '+'. At this point, no node has been parsed and the stack is empty. Thus no node should be retainable. However, Wagner's `process_deletions` only runs *after* parsing has completed, so during error recovery `exists(current_version)` still returns `true` for all nodes. This leads to the subtree E_1 being retained whereas it should be discarded.

implementation of top-down reuse [88, p. 74]. However, top-down reuse is only run *after* parsing has completed, which would mean that this information is not yet available when we try to find retainable subtrees during error recovery (see Figure 3.5 for an example). This suggests that Wagner uses some additional mechanism to determine the existence of nodes during error recovery.

In his thesis, Wagner briefly mentions the node reuse technique by Larchevêque [49]. He writes 'the history mechanisms we define subsume the mark/dispose operations described by Larchevêque' [88, p. 57]. This suggests that Wagner's implementation is based on Larchevêque's, though since the implementation isn't available anymore, this is difficult to verify. The technique involves initially marking nodes in the parse tree as disposable, and then only marking them as reused when they are part of a shift or a reduction. We can assume that this is done lazily, as marking each node in the entire parse tree as disposable *before* re-parsing them, would be needlessly inefficient. With this in mind, the technique can be implemented as follows. Each node has a flag `exists` which we set to `false` whenever that node is encountered during parsing. When the node is pushed onto the parse stack (either via a normal shift, an optimistic shift or a reduction) we set its `exists`-flag to `true`. Using these two steps we can guarantee that the information about whether a node exists in the current parse tree or not is always up to date. Nonterminals

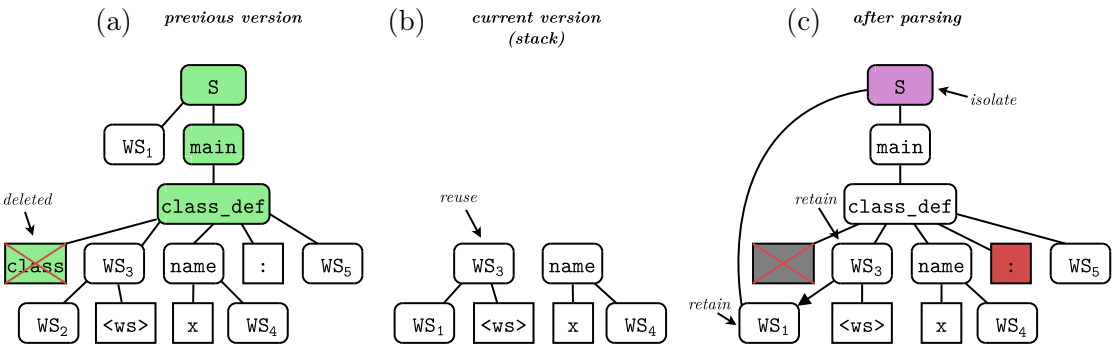


Figure 3.6: An example, showing a bug within the retainability algorithm. **a)** The parse tree after parsing the input '`class x:`' and then deleting the keyword '`class`'. **b)** During re-parsing the node $\texttt{WS}_1$ is optimistically shifted and reassigned to parent $\texttt{WS}_3$, which was reused. **c)** During recovery from the error at '`:`', `S` is isolated which results in $\texttt{WS}_1$ and $\texttt{WS}_3$ being retained. This leads to both `S` and $\texttt{WS}_3$ pointing to $\texttt{WS}_1$ as their child.

that are broken down, but not reused during a reduction, are marked as non-existing as their flag is never set back to `true`. Nodes within optimistically shifted subtrees are not visited and thus keep their flag being set to `true` from the previous parse. There is, however, a small pitfall. Sometimes, optimistically shifted subtrees are undone via right-breakdown when it turns out that shifting them was invalid. This results in the subtree being removed from the parse stack and broken down, which means that the root node and all other nodes that were broken down in the process, need to set their `exists`-flags to `false`. Of course, if during a reduction one of those nodes is reused again and pushed back onto the stack, its `exists` value is set to `true` again.

3.3.5 Retaining empty subtrees

Wagner's algorithm for retaining subtrees within isolations deals with empty nonterminals incorrectly. This section explains why this is a problem and shows how it can be fixed. Consider the following grammar:

```
S:          WS main;
main:       class_def | stmt;
class_def:  "class" WS name ":" WS;
stmt:       name ";";
name:       "ID" WS;
WS:         WS "<ws>" | ;
```

For the input '`class x:`' this grammar produces a valid parse tree. However, when we edit this program by deleting the keyword '`class`', a nonsensical parse tree is produced, as Figure 3.6 shows.

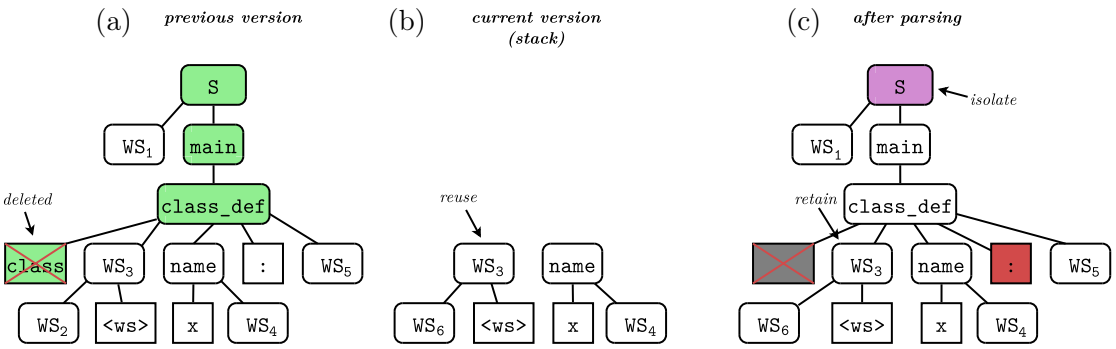


Figure 3.7: An example showing how disallowing nonterminals that don't consume text from being optimistically shifted, solves the problem from Figure 3.6. **a)** The initial parse tree as in Figure 3.6 **b)** Because `WS1` is not shifted onto the stack, a new nonterminal `WS6` needs to be created, which is later reduced to `WS3`. **c)** When `S` is isolated and `WS3` is retained there is no conflict any more, since `WS3` does not reference `WS1`.

In the example, `S` was chosen as the isolation node, because there is no valid position on the stack we can cut back to that allows `main` or `class_def` to be shifted. The problem occurs because the node `WS1` was optimistically shifted and, during its reduction, reassigned to `WS3`, replacing `WS2`. During the search for retainable nodes both `WS1` and `WS3` can be retained, since their text offsets and lengths haven't changed between the previous and current version of the parse tree. However, since `S` was isolated, its changes are being discarded and the node is reset to the previous version. This includes its reference to child `WS1`, whose parent pointer is also reset to reference `S` when it is retained. However, because `WS3` was retained, it still references `WS1`, resulting in two parents referencing the same child.

We can apply a simple fix that solves the part of the problem where two parents reference the same child. However, this still leads to an incorrect parse tree. The function `node.set_parent(`*`parent`*`)` is not explained in Wagner's thesis, though it is obvious that its intention is to set the parent pointer of the node to its new parent. However, it may also remove the node from its previous parent's list of children. Applied to the example in Figure 3.6, this would remove `WS1` from `WS3`'s children, when it is retained and reassigned to `S`. Now `WS1` is only reference by one parent. However, `WS3` is still being retained and is now an invalid subtree since it's missing a `WS` nonterminal (the `WS`-rule requires two children).

We can solve this problem entirely by restricting optimistic shifts in the incremental parser to nonterminals that consume text. The main reason for the problem is that optimistically shifted empty nonterminals can be retained, even if they have been moved into a different subtree. Since their text-length is zero, moving them into another subtree doesn't affect that subtree's text-length and so it can be retained. Thus, not being able to

optimistically shift empty nonterminals circumvents this problem entirely. In the example in Figure 3.6, this solution would keep $\texttt{WS}_1$ from being shifted, and instead being broken down via left breakdown. Since $\texttt{WS}_2$ is also empty, it would also be broken down and a new node $\texttt{WS}_6$ would be created, becoming $\texttt{WS}_3$'s child instead. This allows the algorithm to retain $\texttt{WS}_3$ without any problems (see Figure 3.7).

A small disadvantage of this solution is that it creates new empty nonterminals, throwing away nodes that could have been reused, wasting memory since both the new node and the old node are stored within the parse tree's history. Fortunately, this is not a significant problem in practice, as top-down reuse replaces the newly created nodes again with their equivalents from the previous version of the parse tree before they are stored within the history.

Unfortunately, the above solution also reveals another problem, which happens when: a subtree has no changes in the previous version and is optimistically shifted; due to an error that subtree then needs to be broken down and re-parsed via right-breakdown; the subtree contains an empty nonterminal, which can't be reused because of the solution above; and the subtree is then retained, but its parent is discarded. This then results in a subtree with a change, but no trail of `nested_changes`-flags leading down to it. Because of this those changes won't be processed by top-down reuse and more importantly they won't be logged to the history. See Figure 3.8 for an example.

Luckily, the solution is simple: when retaining a node during the retain-or-discard phase, we check if it contains changes and if so, we mark the parent with the `nested_changes` flag. In the best case, the parent is already marked and there's nothing to do. If not, marking the parent will also mark the parent's parent, and so on, creating a trail of `nested_changes` flags up to the root. This enables top-down reuse and history logging to find and process that change. In the example in Figure 3.8, retaining `A` would mark its parent `B` as having nested changes completing the trail from the root down to the node $\texttt{C}_2$, which can now be found by top-down reuse and replaced with $\texttt{C}_1$; or, if top-down reuse is not utilised, be saved to the history log.

3.4 Out-of-context analysis

For out-of-context analysis, Wagner employs a technique introduced by Petrone [65]. The idea is as follows: to parse a subtree independently from its surrounding context, a new, virtual parse tree is created where the subtree is the entire parse tree. This virtual parse tree is then parsed with a subset of the original grammar that only accepts input that results in the same nonterminal type as the original subtree. This requires

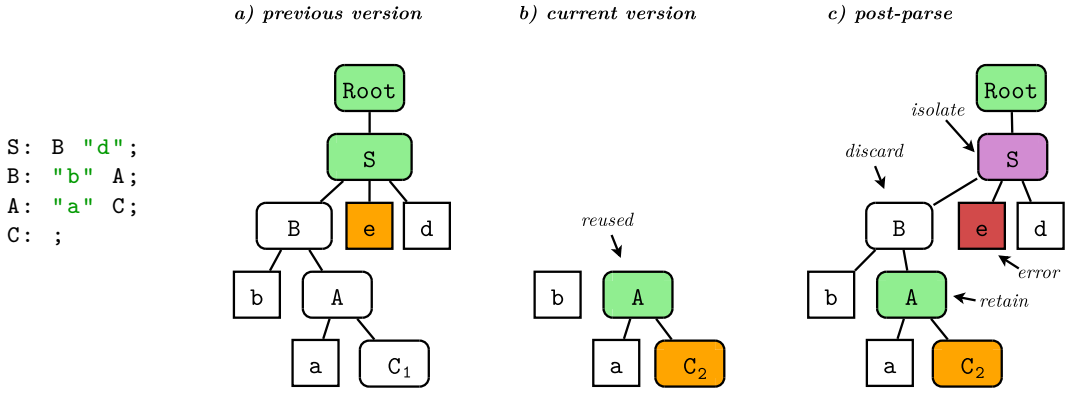


Figure 3.8: a) A parse tree, generated from the input `bd`, that has been edited via insertion of a new node '`e`'. **b)** Initially `B` is optimistically shifted, but needs to be broken down via right-breakdown because it is followed by '`e`' which cannot be shifted. Because of the bug fix from Section 3.3.5, C_1 cannot be reused and needs to be recreated. Then the nodes `a` and C_2 are reduced to `A`, which is reused from the previous version, and its `changed` attribute set to `true`. After shifting '`e`' an error occurs. **c)** During error recovery, `S` is isolated and `B` is discarded since it doesn't exist in the current version. However, `A` can be retained, since it exists in both versions and its offset and length haven't changed. Unfortunately, because `B` has been discarded, there is no trail of `nested_changes` flags leading to node C_2. This means that: it cannot be replaced with its equivalent from the previous version via top-town-reuse; and more importantly, it cannot be reached by the history logging to store the new node C_2. The latter can lead to problems in subsequent parses, since history and current version don't match.

the modification of the grammar to 'allow any symbol to serve as the start symbol' [88, p. 101]. If, after re-parsing the subtree, the symbol on top of the parse stack is the same as the previous root of the subtree, then the re-parsed subtree can be reintegrated into the original parse tree. Otherwise its changes must be discarded again. See Figure 3.9 for an example.

3.4.1 Out-of-context analysis without grammar transformation

One of the problems with this approach is that modifying the grammar adds additional states to the state graph which increases time and memory needed to construct the parse tables. For the grammars tested this lead to on average 50% larger parse tables (see Figure 3.10). The following shows an alternative approach that doesn't require the alteration of the grammar. Instead it modifies the incremental parser to allow it to parse a detached subtree using the original grammar. For this we need to do three things: create a separate incremental parser and initialise it to a state where the subtree can be parsed; create a virtual parse tree where the subtree's preceding and succeeding nodes are used as virtual `bos` and `eos` nodes; modify the parser so that it stops parsing when it reaches the virtual `eos` node. Figure 3.11 shows an example using the same scenario as in Figure 3.9.

(a) Original grammar:

```
E ::= T | E "+" T
T ::= "int" | T "*" "int"
```

New grammar:

N ::= bos_E E eos_E | bos_T T eos_T

```
E ::= T | E "+" T
T ::= "int" | T "*" "int"
```

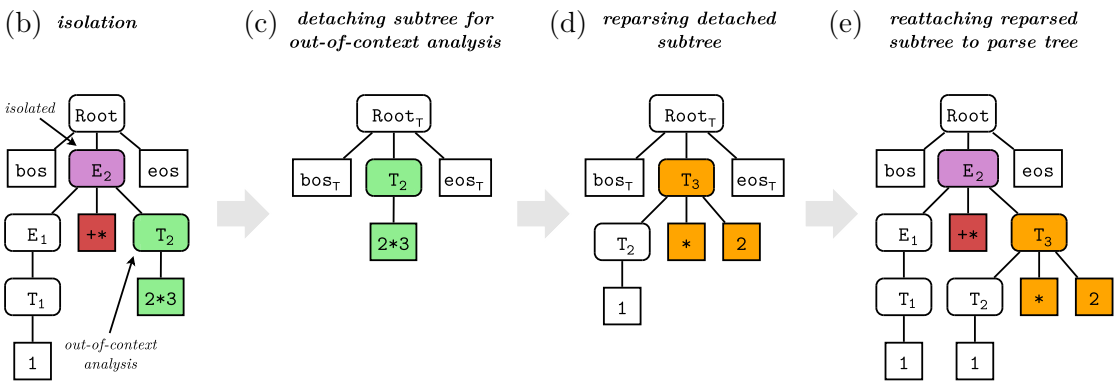


Figure 3.9: An example showing how to independently parse a subtree using out-of-context analysis and reintegrating the results into the parse tree, as described by Wagner. **a)** In order to parse subtrees without their surrounding context, the grammar has been altered as described by Petrone. **b)** Subtree `E` has been isolated, but subtree $\texttt{T}_2$ has changes that can potentially be integrated into the parse tree. **c)** A new virtual parse tree is created with $\texttt{T}_2$ at the top and **d)** reanalysed using a separate parser. **e)** The resulting subtree has the same symbol (`T`) and can thus be reintegrated into the original parse tree.

	Python	PHP	Java
State graph	1.6x	1.2x	1.7x
Parse table	1.8x	1.2x	1.6x

Figure 3.10: A table showing the memory increases of the state graph and parse table when using Petrone's grammar transformation on Python, PHP, and Java grammars to allow using them for out-of-context analysis.

When re-parsing the virtual parse tree with the modified incremental parser, we can't simply return the result as soon as we see the virtual `eos` node. In the example from Figure 3.11, when we see `eos`, the reduction to $\texttt{T}_3$ has not happened yet. If there is more than one element on the parse stack, we thus first need to apply any outstanding reductions using the virtual `eos` node as the lookahead (if that node is a nonterminal, and the parse table hasn't been altered to allow nonterminal lookaheads, we use its most left terminal symbol as the lookahead instead). The out-of-context analysis is successful if the stack has exactly one element, and the parsing state is the same as that of the original parser after parsing the subtree. Afterwards, like Wagner, we check if the symbol of the old subtree and the re-parsed subtree are still the same, and only then reintegrate the changes into the original parse tree. Listing 3.6 shows a simplified algorithm and the changes we need to apply to the incremental parser.

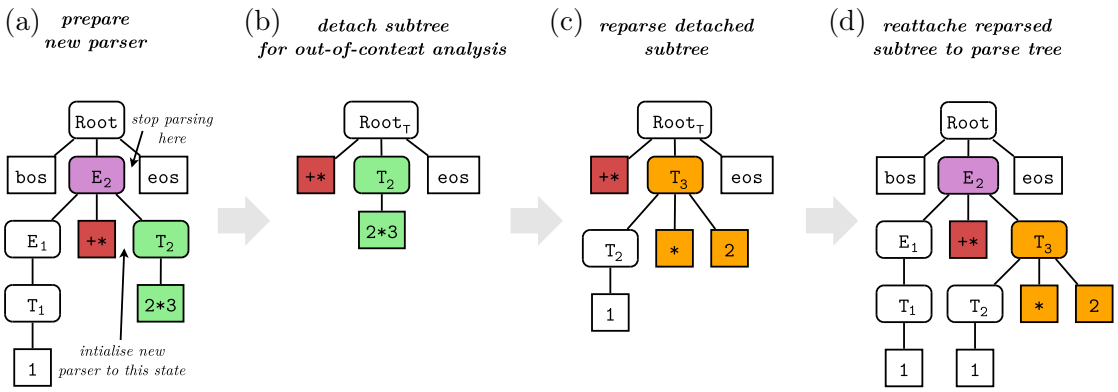


Figure 3.11: An example, showing out-of-context analysis without grammar transformation. The idea is very similar to Petrone's and only requires a few modifications to the incremental parser. **a)** We first initialise a new incremental parser to a state where $\texttt{T}_2$ can be parsed. This is the state just after '+' was pushed during the last successful parse. **b)** We create a new virtual parse tree with the subtree at the top, and the subtree's preceding node as `bos` and its succeeding node as `eos`. **c)** Now the virtual parse tree can be re-parsed by the modified incremental parser. **d)** Like Wagner we compare the re-parsed subtree's symbol with the original subtree and only integrate the result if it is the same.

3.5 Dealing with isolated subtrees during parsing

Once an error has been isolated, the parser can continue re-parsing the rest of the tree as usual. In subsequent parses isolated subtrees are skipped unless they contain new changes or their surrounding context has changed. This section explains some details missing in Wagner's thesis and discusses some problems with isolated subtrees during parsing. Section 3.5.1 explains what the surrounding context of a node is and how it is used to determine if an isolation needs to be re-parsed. Section 3.5.2 shows how to mark isolated subtrees so that they can be found in subsequent parses. Section 3.5.3 describes a problem with the interaction of an isolated subtree and the `right_breakdown` method during parsing. Section 3.5.4 shows an optimisation that increases node reuse during error recovery. Finally, Section 3.5.5 briefly discusses the ways in which we can present errors to the user.

3.5.1 Surrounding context

In his thesis, Wagner describes that during parsing, isolated error locations need to be revisited 'since additional modifications may have changed the surrounding context in such a way that the former error is now valid' [88, p. 94]. However, the only description for surrounding context Wagner offers is the following: 'In general, the mapping between a token and its lexeme is dependent on the surrounding context; a token is lexically

```
1  def out-of-context-analysis_D(subtree: Node):
2    vbos: Node = preceding(subtree)
3    veos: Node = next_lookahead(subtree)
4    pstate: int = subtree.state
5    psymbol: Symbol = subtree.symbol
6    op = new incremental ooc parser
7    op.state = vbos.state
8    op.tree = Root(vbos, subtree, veos>)
9    if op.incparse(veos, pstate)
10       and op.stack[0].symbol == psymbol:
11     integrate changes
12   else:
13     discard changes
```

```
14 class IncrementalOOCParser_D:
15   def incparse_D(veos: Node, target: int):
16     ...
17     if la is veos:
18       while len(stack) > 1:
19         apply reductions using veos
20       if state == target:
21         return True
22       return False
23     ...
```

Listing 3.6: A simplified algorithm for out-of-context analysis without grammar transformation. Before out-of-context analysis of a subtree can commence, we need to create a virtual parse tree (line 8). For that we create virtual `bos` and `eos` nodes using the nodes surrounding the subtree (lines 2–3). We initialise a new (modified) incremental parser and initialise it so it can parse the subtree (line 6-8). When re-parsing the virtual parse tree, we stop immediately when we reach the virtual `eos` node (line 17). If the stack contains multiple elements, this means there is still work to do and we apply any outstanding reductions using the virtual `eos` node as lookahead (lines 18–19). If that node is a nonterminal we can use its most left token instead. Only if the parsing state and the resulting virtual parse tree match the reanalysed subtree's previous state (`pstate`) and symbol (`psymbol`), can the changes be integrated into the original parse tree (lines 9-11, 20). Otherwise they are discarded (line 13).

dependent on the set of tokens that establish sufficient context for determining this mapping'.

The following summarises my understanding of what surrounding context means and gives a simple algorithm for it. Wagner's description, in essence, talks about the relationship between a token and its successor, i.e. the token's lookahead (see Section 2.5.3). From this we can infer that the surrounding context of a nonterminal node also is its lookahead, i.e. the token that was used to lookup the reduction which created that node. If that lookahead changes, then the previous reduction is likely not valid any more and needs to be re-parsed. Figure 3.12 shows an example of surrounding context and gives an algorithm on how to find it. We have already seen the concept of surrounding context before with `right_breakdown` in Section 2.6.2: when we optimistically shift a subtree, we do this without considering its lookahead, which may have changed. The validation phase in the incremental parser thus makes sure that the optimistic shift is reverted, if the lookahead invalidates the previously shifted subtree.

3.5.2 Marking isolation trees and errors

Isolating an error essentially defers the incorporation of changes to a later point in time, when more information has become available to parse those changes successfully. During isolation, the path leading down to those unincorporated changes needs to be marked

```
1 def surrounding_context_Wb(node: Node) -> Node:
2   la: Node = next_lookahead(node)
3   while la is a nonterminal:
4     if la.has_children():
5       la = la.children[0]
6     else:
7       la = next_lookahead(la)
8   return la
```

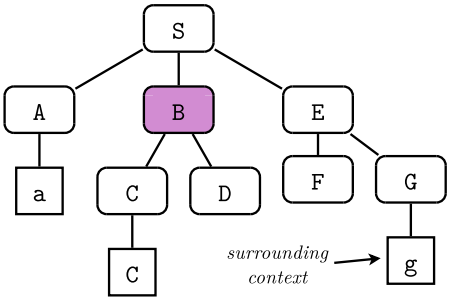


Figure 3.12: To find the surrounding context of a node we traverse the node's successors in the parse tree until we find a token. In the parse tree on the right, the subtree at `B` has been isolated. Its surrounding context is the token `g`, since `g` was used during parsing to determine the reduction of `B`. Thus, if `g` changes, we also have to revisit the isolated subtree as that change may affect the way in which the subtree is parsed.

with `nested_error` flags. This enables the parser to find and re-parse those changes when the isolated subtree is revisited later on. However, the parser only needs to revisit the isolated subtree if it contains new changes or its surrounding context has changed. Wagner describes: 'to locate errors efficiently [...] the path between the root of the tree and each error-containing node must be marked [...] with boolean node annotations similar to the `nested` attribute [...]' [88, p. 94]. This seems to suggest that the entire path from the root down to each node containing an error, is marked. However, later on he writes: 'The paths to unincorporated modifications defined by `nested_error` attributes are terminated immediately below the root of the isolated subtree, preventing subsequent analyses from re-inspecting isolated errors unnecessarily' [88, p. 96]. This suggests that only a part of the path to the error node is marked, however it is unclear whether he means the path from the root to the isolation node, or the path from the isolation node to the error node.

Let's assume that error recovery only marks the path from the isolation node down to the errors, and that there is no path from the parse tree root to the isolation node. Figure 3.13 shows how this behaviour can result in wrong parse trees. The example demonstrates that instead, we need to mark the entire path from the root to the error node in order to find and re-parse isolated changes in subsequent parses: we need a path to the isolation node in order to re-parse it if its surrounding context has changed; but we also need a path from the isolation node to all the isolated changes, so that when the isolated subtree is being re-parsed, we can find those changes and reanalyse them. However, like Wagner we don't want to re-parse isolated subtrees unless doing so leads to new changes being incorporated into the parse tree. We can do this by leaving the isolation node itself unmarked and only re-parse it if it contains new user modifications or its surrounding context has changed. Otherwise, the isolated subtree is skipped by optimistically shifting it, as though it is a subtree without any changes. Figure 3.14 shows how the problem

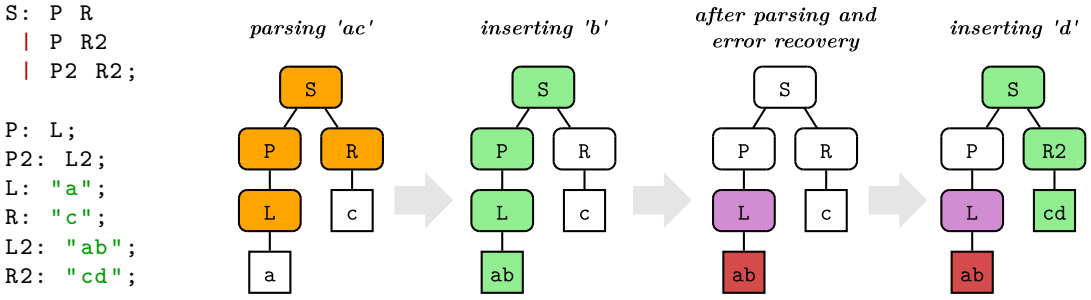


Figure 3.13: An example showing why only marking error paths within the isolation subtree is insufficient. Using the grammar on the left, the input `ac` is parsed and then edited by first inserting '`b`' (after '`a`') and then '`d`' (after '`c`'). After inserting '`d`', the isolated subtree `L` needs to be re-parsed, since its surrounding context was changed. However, there is no `nested_errors` path leading down to `L`, and since `P` has no nested changes, it is optimistically shifted by the parser. This then leads to a wrong parse tree, since despite the input `abcd` being valid, the parser contains isolated errors, and `P` and `L` should have been re-parsed as `P2` and `L2`, respectively.

from Figure 3.13 is solved when we assume that error nodes are marked from the root down.

To mark isolated changes as errors during error recovery, Wagner uses the function `discard_and_mark_errors` (shown in Listing 3.7 on the left). Its description contains the following: 'Any user modifications (textual or structural) within this subtree are marked as unincorporated errors, and nested error attributes are set to record the path between node and the location of each such error' [88, p.99]. The description suggests that any node that is unretainable and contains nested changes is marked with a `nested_error` flag. However, the code only sets a node's `nested_error` flag if one of its children has local or nested errors. But since the function is called when traversing the tree from top to bottom, the children are marked *after* their parent. This would mean that when a parent checks if any of its children has errors, the children can't possibly have been marked yet. We thus re-implement this function and simply set a node's `nested_errors` flag if it has nested changes as shown in Listing 3.7 on the right.

3.5.3 Right-breakdown problem

In certain scenarios, isolated subtrees can cause a problem with the incremental parser's right-breakdown procedure. The problem occurs when, during parsing, an isolated subtree is followed by an empty nonterminal, which was optimistically shifted, and another error is found afterwards. We recall that after an optimistic shift the parser enters a verification phase. The phase ends when a terminal symbol is shifted. If an error occurs during it, the method `right_breakdown` is called to undo the optimistic shift and re-evaluate the subtree's contents. This is done by popping the subtree from the top of the stack and

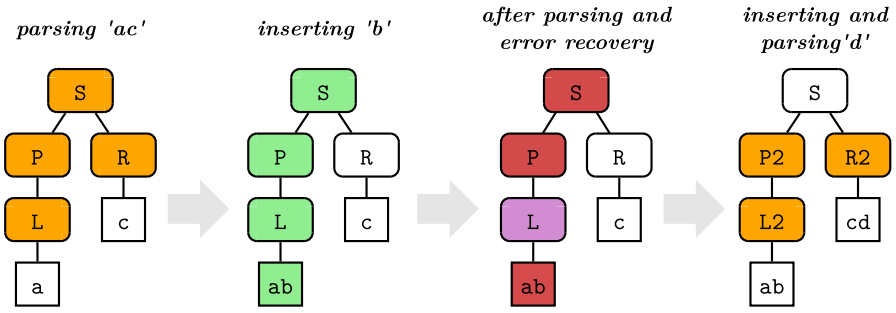


Figure 3.14: An example showing how also having a path of `nested_errors` flags from the root down to the isolation node, enables the parser to find the isolated subtree during a re-parse. However, the parser only re-analyses the isolated subtree if it has new changes, or if its surrounding context was changed. In the example, inserting '`d`' changes the surrounding context of isolated subtree `L`. When the incremental parser reaches the node `L`, it thus re-inspects the subtree and re-parses all previously isolated changes (i.e. nodes previously marked as errors). This allows the parser to correctly parse the scenario from Figure 3.13.

breaking down its most right edge, shifting the children back onto the stack. As long as there is a nonterminal on top of the stack this process is repeated until right-breakdown can shift a terminal symbol. However, when an isolated subtree is broken down this way, and it contains an error node in its most right edge, then the right-breakdown procedure fails, as it assumes that the subtree only contains valid input. Figure 3.15 illustrates the problematic scenario.

One way of solving this is to use the fix of Section 3.3.5, which also happens to solve this problem. However, a second solution is possible[3]: if during `right_breakdown` an isolated subtree ends up on the top of the stack, the routine is simply aborted and the verification phase ends. This means that the isolated subtree is not broken down and remains on the stack, and parsing just continues as normal. Since the parser couldn't use right-breakdown during the verification phase, it is likely that the error remains. However, this just results in another isolation, e.g. in Figure 3.15 node `S` would be isolated as well.

Since this problem completely disappears when using the fix from Section 3.3.5 – if empty nonterminals cannot be optimistically shifted, isolated subtrees cannot end up in the verification phase – there is no need to use the solution presented here. However, if the problem from Section 3.3.5 can be solved without the need to disallow optimistic shifts of empty nonterminals, then this problem would remain, and the solution presented here needs to be applied.

[3]This solution has been discussed with Wagner in a personal communication and was described by him as 'reasonable' [87].

```
def discard_and_mark_errors_W(node: Node):
  node.discard()
  if node.has_changes(reference_version, local):
    if not node.local_errors:
      node.local_errors = True
      node.compute_presentation(reference_version)
  if ∃child of node s.t.
      (child.local_errors or child.nested_errors):
    node.nested_errors = True
  else:
    node.nested_errors = False
```

```
def discard_and_mark_D(node: Node):
  node.load(previous version)
  if node.changed:
    node.local_error = True
  if node.nested_changes:
    node.nested_errors = True
  else:
    node.nested_errors = False
```

Listing 3.7: Wagner's original function (converted to Python) for discarding changes and marking errors (left) and our re-implementation (right). The main difference between the two functions is the way we mark nested errors. Since, when a node is discarded we haven't processed its children yet, we can't possibly know whether any of the children contain errors. Instead we can use the `nested_changes` attributes to mark errors. Like Wagner we consider any node within an isolation that has unparsed changes, as an error. Thus any subtree that has nested changes, will have nested errors once isolation is done. We can thus simply set the `nested_errors` flag of any node that has nested changes.

3.5.4 Node reuse and error recovery

Section 2.8 described that during a re-parse, any node that is being reused needs to be remembered so that it can not be reused twice. When cutting back the stack to find an isolation node, we remove nodes from the parse stack again. At this point however, nodes will already have been marked as reused, which means they cannot be reused again and need to be recreated when parsing continues (see Figure 3.16 for an example). However, removing nodes from the stack during error recovery is essentially like pretending that they have never been parsed, and thus they shouldn't need to be recreated afterwards.

There are two solutions to this problem. The obvious and most effective solution is to simply remove all nodes from the reuse set that have been removed from the stack during the search for an isolation node. This allows those nodes to be reused again when parsing continues. However, this may break the modularity of incremental parsing and error recovery, since we need to make the reuse set available during error recovery. An alternative solution is thus to simply ignore the problem, and let top-down reuse take care of the new nodes. After parsing has finished, it will automatically find the recreated nodes and replace them with their equivalent from the previous version. However, this carries a small overhead of creating new nodes only to immediately destroy and replace them again. To mitigate this problem we can also modify `ambig_node_reuse` from Listing 2.11 to only add nodes to the reuse set if they have more than one child (see Listing 3.8). The purpose of the reuse set is to keep parent nodes from being reused twice during parsing, which can only happen if they have multiple children. Not adding parents with a single

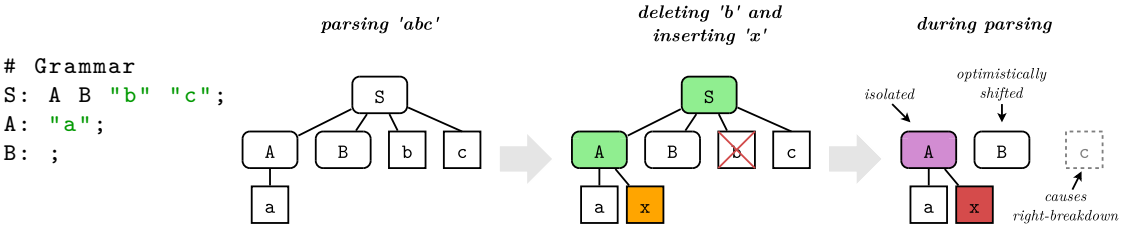


Figure 3.15: An example showing the interaction between right-breakdown and an isolated subtree causing a bug in the routine. A parse tree has been generated and edited in a way that causes the subtree `A` to be isolated. After the isolation the parser continues parsing. Empty subtree `B` has no changes and can thus be optimistically shifted, and the parser enters the verification phase. However, when the lookahead '`c`' is processed, a parsing error occurs, and right-breakdown is used to undo the optimistic shift. This leads to the isolation subtree being broken down and its children being shifted onto the stack. This includes the child '`x`', which is not a valid input. However, since right-breakdown assumes all input to be valid, this causes a runtime error.

child to the reuse set is thus a valid option and will allow such parent nodes to be reused again even if they have been removed from the stack during error recovery.

3.5.5 Displaying errors

Once an error has been isolated, we need to show to the user what caused the error and give them hints on how they can fix the problem. In his thesis, Wagner describes how the isolation can be used to precisely highlight the user changes that led to the error, by simply marking all changed but unparsed tokens within an isolated subtree as errors [88, p. 103]. For example, if the user inserted or deleted a token which caused a parsing error, then Wagner shows them the changed token with a hint that removing or re-entering that token may fix the problem. Wagner's way of presenting errors can be very useful when the actual parse error happens far away from the change that caused it, in which case other error recovery approaches can sometimes give unhelpful or misleading information about the source of the error [59, 76, 20]. On the other hand, simply pointing to the user change as the source of the error isn't always useful either, especially if the changes were intentional, as Figure 3.17 shows. Instead of relying on only one representation, we can combine both approaches, so that the user receives all available information to understand why an error occurred. Parse errors are highlighted with a red squiggly line, and the change that caused the error is highlighted with a blue one (see Figure 3.18).

To display errors in the editor Wagner marks all user changes that couldn't be retained as errors using `local_error` flags and computes how those errors should be represented. Since we also want to highlight the actual parse error, we need to mark the node that caused the error as well. Once an error is fixed, marked error nodes need to be unmarked again, so that the editor stops rendering them as errors. This can simply be done by

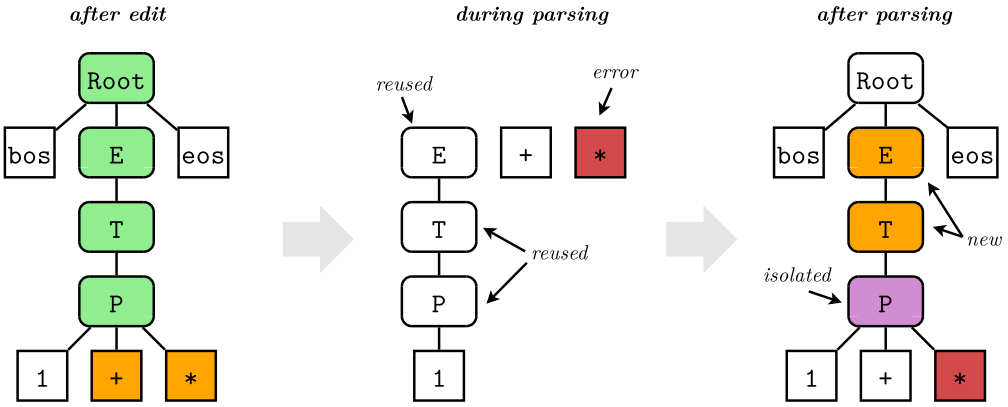


Figure 3.16: An example showing how error recovery can result in nodes not being properly reused. During re-parsing of the user edits, the incremental parser can reuse the nodes `P`, `T`, and `E` before reaching a parsing error. This means that those nodes have been marked as reused and cannot be reused again. During error recovery however, the nodes are popped from the stack again in search of an isolation node, which is found with `P`. After isolating `P`, the parser continues as usual. When reducing `P` to the symbol `T` we would normally reuse previous node `T`. Unfortunately, it has already been marked as reused, so the parser needs to create a new node instead.

```
1 def ambig_reuse_check_WD(prod: Symbol, children: List[Node]) -> Node:
2   for c in children:
3     if c is not a new node:
4       old_parent: Node = c.get_parent(previous version)
5       if old_parent.symbol == prod and old_parent not in reused list:
6         if len(old_parent.get_children(previous version)) > 1:
7           add old_parent to reused list
8         return old_parent
9   return Node(prod, children)
```

Listing 3.8: An updated function for ambiguous node reuse. Since only nodes with multiple children can run the danger of being reused twice, we do not have to remember nodes with single or no children (line 6). This allows us to reuse the same node again, even if it was previously discarded during error recovery.

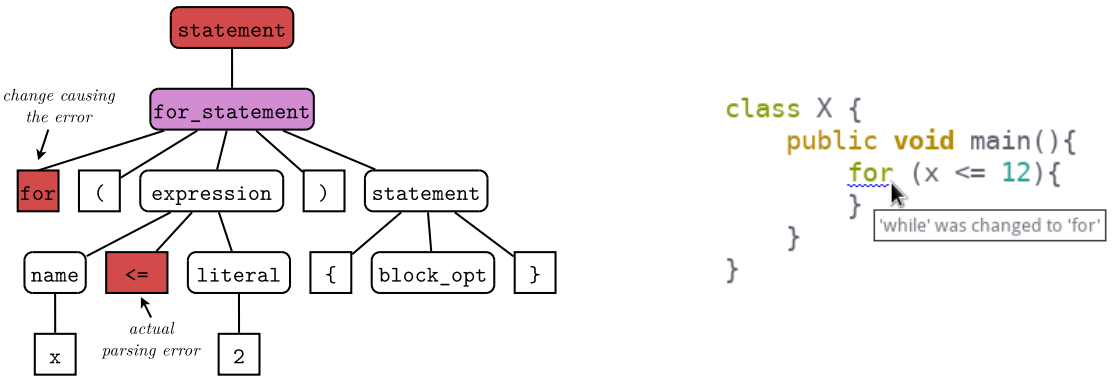


Figure 3.17: An example showing that highlighting only the user changes when an error occurs isn't always sufficient to help the user understand the problem. Here, the user intended to replace a while-loop with a for-loop. However, editing only the keyword is not enough, since the condition also needs to be changed. The change thus causes an error when the parser tries to parse '`<=`' (a for-loop requires the condition to start with an assignment). The isolated subtree is shown on the left, the error reporting is displayed on the right. Simply highlighting that the change from `while` to `for` caused the error doesn't help the user understand what went wrong. A more helpful error message, in this case, would be to tell the user that the problem lies within the condition.

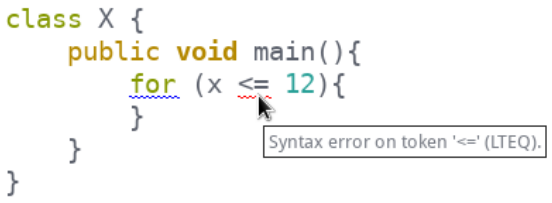


Figure 3.18: An example of an error being presented using both Wagner's as well as a traditional approach. The change responsible for the parsing error is shown with a blue squiggle. The actual parsing error is shown with a red squiggle. Hovering over the `for`, tells the user that changing the `while` is the cause of the error, while hovering over '`<=`' shows more information about the nature of the error.

unmarking a node when it is successfully shifted during a re-parse. However, often parse errors can be resolved by changing the nodes preceding the error node, which means the error node itself is never re-parsed. For instance, the error in Figure 3.17 can be resolved by simply changing back the `for` to a `while`. The subtree `expression`, which contains the marked error node, would be optimistically shifted since it has no changes. This means the error node '`<=`' is not visited again by the parser and thus can't be unmarked. To solve this, we simply store a reference to the actual error node on the isolation node. When the isolation node is revisited during a re-parse, we unmark the referenced error node, assuming it will be fixed. If the error remains, the node will be remarked again by the error recovery. If the error is indeed resolved, the node has been unmarked and won't be rendered as an error any more.

Wagner's algorithm marks nodes within an isolation as errors, if they can't be retained. However, since tokens cannot change their text-length or offset between the previous and current version, they are always retainable. This means, they won't be marked as errors

and thus won't be highlighted in the editor. For example, in Figure 3.17 the changed node `for` is retainable and wouldn't be marked as an error. We thus simply assume that tokens are unretainable by default, which then allows them to be marked and displayed as errors, if they contain changes that could not be successfully parsed.

Chapter 4

Incremental Parsing Extensions

Using incremental parsing and having access to an always up-to-date parse tree allows an editor to return almost instant feedback of syntax errors and makes implementing features such as syntax-highlighting effortless. Other common IDE features, however, require a more condensed form of the parse tree, an abstract syntax tree (AST), to work with, such as name binding, which provides an IDE with information about undefined or unused variables and enables context-aware code completion. Section 4.1 of this chapter thus explains how we can extend an incremental parser to generated and update ASTs incrementally during parsing, and storing them along with the parse tree.

Despite Wagner's incremental parsing algorithm being language agnostic and working well for most standard languages, there are some languages that require additional work prior to parsing. Such a language is Python, whose grammar requires indentation tokens to be inserted into the program to denote the beginning and ending of blocks. Typically, this is either done during the lexing phase, or as a separate phase between lexing and parsing. Unfortunately, this requires the revisiting of the entire program in order to insert the correct amount of indentation tokens, as a change of indentation at the beginning of the program can invalidate indentations further down. While we can avoid unnecessary parse tree changes by only updating indentation tokens in the parse tree if they were changed, we are still required to scan the file from the top to the bottom. Section 4.2 thus shows how we can incrementally find and update only those lines in the program whose indentation has changed, without the need to analyse the entire program. This is then added as an additional phase between lexing and parsing, which repairs the parse tree back to a state where it can be successfully parsed by the incremental parser.

4.1 Incremental Abstract Syntax Trees

Parse trees are painful to work with: their nesting is partly dictated by the LR parser, and is often very deep; they contain irrelevant tokens, which are necessary only for the parser or to make the language more visually appealing to users; and child nodes are ordered and only accessible via numeric indices. Instead, one would prefer to work with an AST, providing a simplified view on the user's data, where the tree has been flattened as much as possible, with irrelevant tokens removed, and with child nodes unordered and addressable by name.

This section first describes a simple (relatively standard) rewriting language that can be used to create ASTs from parse trees. It then describes the novel technique that was developed to make AST updates incremental.

4.1.1 Rewriting language

The simple rewriting language to create ASTs from parse trees is in the vein of similar languages such as TXL [18] and Stratego [12]. In essence, it is a pure functional language which takes parse tree nodes as input and produces AST nodes as output. Each production rule in a grammar can optionally define a single rewrite rule. AST nodes have a name, and zero or more unordered, explicitly named, children. The AST is, in effect, dynamically typed and implicitly defined by the rewrite rules[1].

An elided example from the Python grammar is as follows:

```
1  print_stmt : "PRINT"              {Print(stmts=[])}
2             | "PRINT" stmt_loop;   {Print(stmts=#1)}
3
4  stmt_loop  : stmt_loop stmt       {#0 + [#1]}
5             | stmt;                {[#0]}
6
7  stmt       : expr                 {#0}
8             | ...
9
10 expr       : "VAR"                {Var(name=#0)}
11            | ...
```

AST constructors are akin to function calls. Expressions of the form `#n` take the nth child from the nonterminal that results from a grammar's production rule. Referencing a token uses it as-is in the AST (e.g. line 10); referencing a nonterminal uses the AST subtree that the nonterminal points to. For example, `Var(name=#0)` means "create an AST

[1] This is not an important design decision; the AST could be statically typed.

```
1  def reduce_D(la: Node, p: Production):
2    ...
3    if p.has_rewriterule():
4      exec_rewriterule(la, p)
5
6  def exec_rewriterule_D(node: Node, p: Production):
7    astnode: AstNode = p.rewriterule.execute(node)
8    if not is_reusable_astnode(node.ast, astnode):
9      node.ast = astnode
10
11 def is_reusable_astnode_D(old: AstNode, new: AstNode):
12   if old is None:
13     return False
14   if old.name != new.name:
15     return False
16   if children are not the same:
17     return False
18   return True
```

Listing 4.1: During the reduction of a node using the production `prod`, we check if the production has a rewrite rule (Lines 3–4). If a rewrite rule exists, it is executed which results in a new AST node, which is then added as a reference `ast` to the result of the reduction, `node` (Line 7–9). If a node was reused during a reduction, we can also check if the AST node is reusable, in which case it doesn't need to be replaced (Lines 11–18).

element named `Var` with an edge `name` which points to a `VAR` token" and `Print(stmts=#1)` means "create an AST element named `Print` with an edge `stmts` which points to the AST constructed from the `stmt_loop` production rule". A common idiom is to flatten a recursive rule (forced on the grammar author by the very nature of LR grammars) into a list of elements (lines 4 and 5). Note that a rewrite rule can produce more than one AST node (e.g. line 1 produces both a `Print` node and an empty list node).

4.1.2 Incremental ASTs

All previous approaches of which I am aware either batch create ASTs from parse trees or use attribute grammars to perform calculations as parsing is performed (e.g. [11]). In this subsection, I explain how Wagner's incremental parser can be easily extended to incrementally create ASTs (the changes are shown in Listing 4.1).

The mechanism adds a new attribute `ast` to nonterminals in the parse tree. Every `ast` attribute references a corresponding AST node. The AST in turn uses direct references to tokens in the parse tree. In other words, the AST is a separate tree from the parse tree, except that it shares tokens directly with the parse tree. Sharing tokens between the parse tree and the AST is the key to this approach since it means that changes to a token's value automatically update the AST without further calculation. Altering the incremental parser to detect changes to tokens would be far more complex.

In all other cases, we rely on a simple modification to the incremental parser. Nonterminals are created by the parser when it reduces one or more elements from its stack. Every altered subtree is guaranteed to be re-parsed and changed subtrees will either lead to fresh nonterminals being created or previous identical nonterminals being reused. We can therefore add to the parser's reduction step an execution of the corresponding production rule's rewrite rule; the result of that execution then forms the `ast` reference of the newly created or reused nonterminal. Note, that like the parse tree, the AST can also reuse AST nodes from previous parses. Though this doesn't happen automatically, it can be easily added, by comparing the resulting AST node from the rewrite rule's execution to the `ast` reference stored on the nonterminal. If they are equivalent, the previous AST node can be reused.

We then rely on two properties that hold between the parse tree and ASTs. First, the AST only consists of nodes that were created from the parse tree (i.e. we do not have to worry about disconnected trees within the AST). Second, the rewrite language cannot create references from child to parent nodes in the AST. With these two properties, we can then guarantee that the AST is always correct with respect to the parse tree, since the incremental parser itself updates the AST at the same time as the parse tree. Figure 4.1 shows this process in action.

This approach is easy to implement and also inherits Wagner's optimality guarantees: it is guaranteed that we update only the minimal number of nodes necessary to ensure the parse tree and AST are in sync.

4.2 Indentation-based languages

Indentation-based languages such as Python require more support than a traditional lexer and parser offer. Augmenting batch-orientated approaches with such support is relatively simple, but, to the best of my knowledge, no-one has successfully augmented an incremental parser before. This section therefore describes how we can extend an incremental parser to deal with indentation-based languages.

The basic problem can be seen in this simplified Python grammar fragment:

```
if    ::= "IF" expr ":" suite;
while ::= "WHILE" expr ":" suite;
suite ::= "NEWLINE" "INDENT" stmts "DEDENT";
stmts ::= stmts stmt "NEWLINE"
        | stmt "NEWLINE";
```

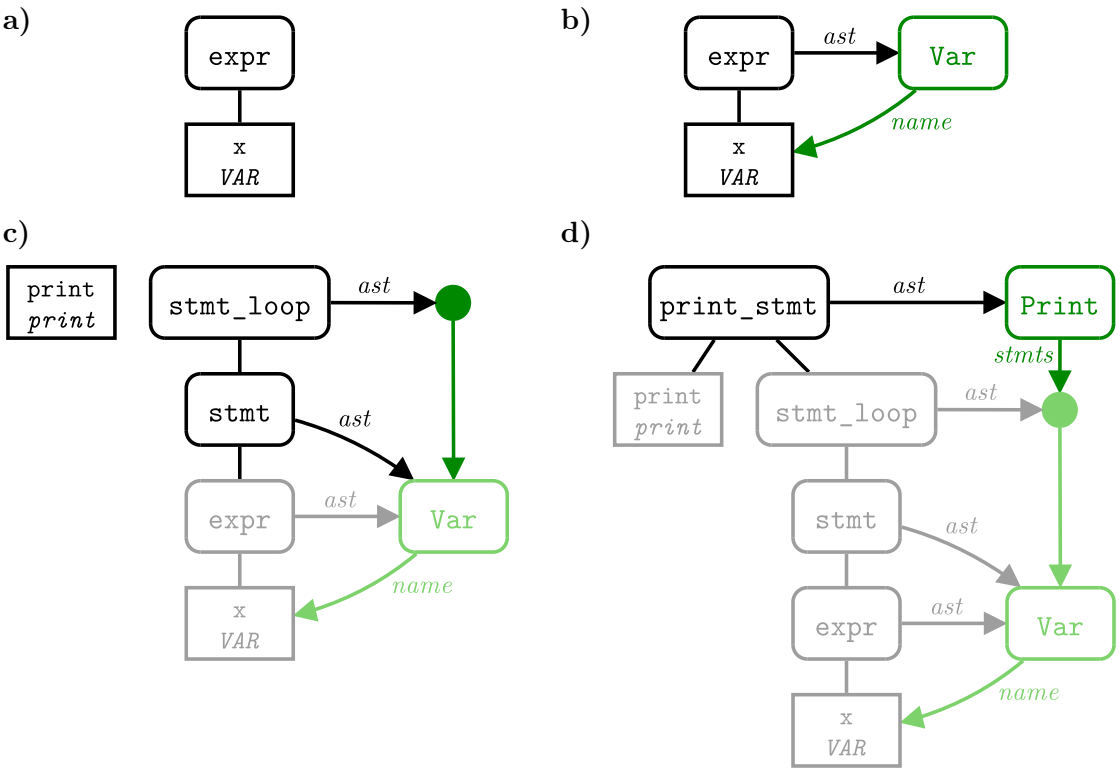


Figure 4.1: Incremental AST construction, with the parse tree shown in black and the AST in green. Subtrees that have been reused are in grey / light green. **a)** After typing the input `x`, the incremental parser creates this parse tree fragment. **b)** After the `expr` nonterminal is created, the rewrite language is run on it, creating an `ast` reference to an AST node `Var`. **c)** After changing the input to `print x`, the incremental parser starts to update the parse tree and the associated AST as shown in this in-process fragment. The `stmt` production's rewrite rule simply references whatever AST node its child produces, so `stmt`'s `ast` reference is the existing `Var` node. `stmt_loop` however wraps its contents in a list (the green circle). **d)** The final parse tree and AST. The `print` production rule creates a `Print` AST element with a child `stmts` which is a list containing a `Var` node.

This grammar contains three special tokens `NEWLINE`, `INDENT`, and `DEDENT` which are equivalent to the semicolon and opening/closing braces of a language like Java. They are not directly specified by the user, but instead have to be generated by the parsing system. A common way to do this is to insert a separate step between the lexing and parsing phase. Other implementations may choose to integrate the insertion of the special tokens into either the lexer or the parser.

As an example, let's consider the following Python program:

```
a = 3
while True:
  a -= 1
  if a == 0:
    break
print a
```

We can see that with the lack of curly braces, the only indication that shows which lines make up a block is their level of indentation. Thus, we cannot simply consume all whitespace during lexing, as in most languages. Instead, line 3 should generate `NEWLINE` and `INDENT` tokens before the '`a`' token (as should line 5, before the '`break`' token), and (because of nesting) two `DEDENT` tokens must be added after the '`break`'. All other lines need to end with a `NEWLINE` token. To parse this program successfully indentation tokens need to be inserted as shown below:

```
a = 3 NEWLINE
while True: NEWLINE
  INDENT a -= 1 NEWLINE
  if a == 0: NEWLINE
    INDENT break NEWLINE
DEDENT DEDENT print a NEWLINE
```

Note that indentation related tokens are solely for the parser's benefit and do not affect rendering: whitespace is recorded as per Section 2.6.3 and rendered as normal.

4.2.1 Indentation in a batch parser

The following shows an example for an indentation algorithm in a batch parser environment, which would run as an additional phase between lexing and parsing (the algorithm assumes that empty or comment-only lines have been removed):

```
1  indentl = [0]
2  for l in source.lines:
3    ws = get_leading_ws(l)
4    if ws > indentl[-1]:
5      l.insert(0, "INDENT")
6      indentl.append(ws)
7    elif ws < indentl[-1]:
8      while ws < indentl[-1]:
9        indentl.pop()
10       l.insert(0, "DEDENT")
11     if ws > indentl[-1]:
12       raise UnbalancedError()
13   l.insert(-1, "NEWLINE")
```

The algorithm first initialises the indentation level stack `indentl` with 0 (line 1). Afterwards it traverses the source code one line at a time (line 2). For each line it counts the leading whitespace (line 3) and checks which one of two conditions is true. If the whitespace is bigger than the last indentation level, the current line is indented so it generates and inserts one `INDENT` token into the source code (line 4–5). It also appends the current whitespace to the indentation level stack (line 6). If the whitespace is smaller, the current line has been dedented so it needs to generate and insert one or more `DEDENT` tokens, depending on how deeply nested the previous line was. For this it simply pops

indentation levels from the stack and generates a `DEDENT` token each time until the indentation level on the stack matches the whitespace in the current line (line 8–10). If there is no match on the stack, the indentation is unbalanced and the algorithm returns an error (lines 11–12). If the whitespace of the current line is the same as the top of the indentation level stack, this means that the current line has the same indentation as its predecessor which requires no additional tokens, apart from a `NEWLINE` token, which the algorithm always inserts at the end of each line (line 13).

Unfortunately, generating indentation tokens using the traditional approach is not suitable for an incremental parser since the algorithm can only generate indentation tokens for the entire file. This would lead to so many changes, that effectively the entire parse tree needs to be re-parsed. A partial solution to this is to only insert indentation tokens in those lines where the existing indentation tokens don't match anymore. However, the algorithm would still require the analysis of every line in the program, which could lead to slowdowns in the editor.

4.2.2 Incrementally handling indentation

The following explains how we can make a batch-orientated algorithm as shown in the previous section, incremental. Similar to that algorithm, the incremental version runs as an additional phase between the lexer and the parser. However, while the batch-orientated version iterates over all lines of the program and inserts indentation tokens as needed, the incremental version needs to be capable of independently updating only partial lines of the program, removing existing indentation tokens or inserting new ones as appropriate. Since the indentation of a line typically depends on a previous line, an incremental version also needs to find all dependent lines of a change and update them as well.

To make this possible, we store in each line the leading whitespace level (i.e. the number of space characters) and the indentation level. These notions are separated, because the same indentation level in two disconnected parts of a file may relate to different leading whitespace levels (e.g. in one `if` statement, 2 space characters may constitute a single indentation level; in another, 4 space characters may constitute a single indentation level). For example, the following is valid Python:

```
if x:
  y
if a:
    b
```

However, the following fragment is *unbalanced* (i.e. the file's indentation is nonsensical) and should be flagged as a syntax error:

```
1  def calc_indentl_D(l: Line):
2    l.ws = get_leading_ws(l)
3    if prev(l) == None:
4      l.indentl = 0
5    elif prev(l).ws == l.ws:
6      l.indentl = prev(l).indentl
7    elif prev(l).ws < l.ws:
8      l.indentl = prev(l).indentl + 1
9    else:
10     assert prev(l).ws > l.ws
11     prevl: Line = prev(prev(l))
12     while prevl != None:
13       if prevl.ws == l.ws:
14         l.indentl = prevl.indentl
15         return
16       elif prevl.ws < l.ws:
17         break
18       prevl = prev(prevl)
19     mark_unbalanced(l)
```

Listing 4.2: The indentation level calculation algorithm taking a line l as input. There are 4 cases, the first 3 of which are trivial, though the last is more subtle: **(1)** If l is the first line in the file its indentation level is set to 0 (lines 3–4). **(2)** If l's whitespace level is the same as the previous line then l is part of the same block and should have the same indentation level (lines 5–6). **(3)** If l's whitespace is bigger, then l opens a new block and has an indentation level 1 more than the preceding line (lines 7–8). **(4)** If l's whitespace is smaller, then either l closes a (possibly multi-level) block or the overall file has become unbalanced (lines 9–19). To determine this we have to search backwards to find a line with the same leading whitespace level as l. If we find such a line, we set l's indentation level to that line's level (lines 12–14). If no such line is found, or if we encounter a line with a lower leading whitespace level (lines 16–17), then the file is unbalanced and we need to mark the line as such (line 19) to force the editor to display an error at that point in the file.

```
if x:
    a
  b
```

When a line l is updated, there are two cases. If l's leading whitespace level has not changed, no further recalculations are needed. In all other cases, the indentation level of l, and all lines that depend on it, must be recalculated; indentation related tokens must then be added or removed to each line as needed. Dependent lines are all non-empty[2] lines after l up to, and including, the first line whose leading whitespace level is less than that of l, or to the end of the file, if no such line exists.

We can define a simple algorithm to calculate the indentation level of an individual line l. We first define every line to have attributes `ws`, its leading whitespace, and `indentl`, its indentation level. `prev(l)` returns the first non-empty predecessor line of l in the file, returning `None` when no such line exists. The algorithm is shown in Listing 4.2. In

[2]Note that comment-only lines also count as empty lines

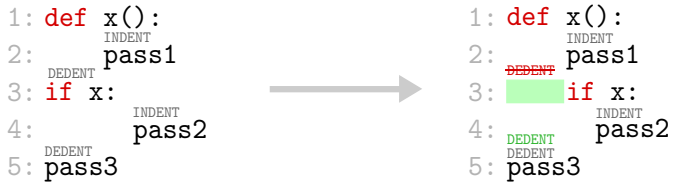
```
def update_dependent_linesD(l: Line):
  while True:
    next_l: Line = next(l)
    if next_l is None:
      break
    calc_indentl(next_l)
    if nextl.ws < l.ws:
      break
```

Listing 4.3: Algorithm to update dependent lines. After the indentation level of a changed line was updated, we need to check and potentially update all lines that are dependent on that change, which are all lines after l up to, and including, the first line whose leading whitespace level is less than that of l, or to the end of the file, if no such line exists.

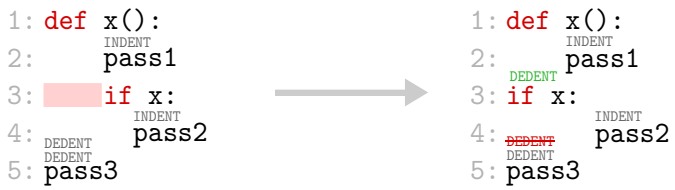
practice, this algorithm tends to check only a small number of preceding lines (often only 1). The worst cases, e.g. an unbalanced file where the last line is modified and all preceding lines need to be checked, are $O(n)$ (where n is the number of lines in the file).

Each time a line has been affected by this process, we need to check whether the indentation tokens in the parse tree match the line's current state. If they do not, the tokens in the parse tree need to be updated appropriately (i.e. the old tokens are removed and replaced). If a line is marked as unbalanced, a single `UNBALANCED` token is inserted; otherwise we insert `INDENT` / `DEDENT` tokens according to the indentation level difference between the line and its first non-blank predecessor.

Once a line has been updated, we need to check if any of its dependent lines need updating as well, which we can do with the algorithm shown in Listing 4.3. The following example shows a scenario where an increase in indentation requires a succeeding line to be updated as well (`NEWLINE` tokens have been elided from the example):



Increasing the whitespace before the `if` updates line 3 by removing the `DEDENT` token before the if-statement. Since line 5 is the first line after line 3 with a smaller whitespace level, it is a dependent line and thus needs to be updated, which adds an additional `DEDENT` token before `pass3`. The next example shows the reverse of the previous one, where the decrease of whitespace requires one of the following lines to be updated:



The deletion of the whitespace before the if-statement in line 3, requires the insertion of a `DEDENT` token at the beginning of that line. As before, line 5 is dependent on that change and needs to be updated by removing one of the `DEDENT` tokens at the beginning.

Once all changed and dependent lines have been updated, added or removed indentation tokens are marked as changed, similar to user changes. We then rely on the incremental parser to reorder the tree appropriately and flag any `UNBALANCED` tokens as parsing errors.

4.2.3 Related work

Erdweg et al. propose a solution for generating layout-sensitive parsers using layout constraints [24]. Using layout constraints, language grammars can enforce indentation and alignment rules within the source code by defining shapes via annotating productions in the grammar. These rules are then applied on the parse forest after parsing, removing all parse trees which do not satisfy the constraints, thus disambiguating the input. Unfortunately, layout constraints come with a significant performance penalty, e.g. leading to parsing overheads of 80% on average when disambiguating Haskell programs.

Amorim et al. propose layout declarations based on layout constraints, improving usability and performance, and adding the option to derive a pretty-printer from the specification [2]. Similar to layout constraints, layout declarations require annotating production rules within a grammar, though defining them is less verbose according to [2, p. 5]. Whereas layout constraints perform post-parse disambiguation, layout declarations disambiguate at parse-time. To do this, nodes are annotated with position information during construction of the parse tree. Each time a production is reduced that has annotated layout declarations, those constraints are checked against the children of the production. If this check fails, the parse tree is rejected and removed from the parse forest. While layout declarations are a clever way of defining and enforcing layout-sensitive languages, it is not clear how this technique can be applied to an incremental non-generalised parser, which doesn't have access to a parse forest containing all interpretations of a program. For example, in an LR parser the decision as to whether a statement belongs to a certain block needs to be made before that statement is parsed, as depending on this, the block is reduced before or after the statement. In order to parse a language like Python, it is thus necessary to insert indentation tokens into the parse tree before the parsing phase is initiated.

Chapter 5

Editing composed languages using language boxes

Language composition needs an editing approach which can combine SDE's flexible and reliable approach to constructing ASTs with the 'feel' of traditional text editing. In part due to MPS's gradual evolution from a pure SDE to an approach which partially resembles parsing, the idea is to do the opposite and start from a parsing perspective and try to move towards SDE. Doing so implicitly rules out any approach which can accept ambiguous grammars. Since the largest class of unambiguous grammars we can precisely define is the LR(k) grammars [44] they were the obvious starting point.[1] The following sections show how an incremental parser which accepts LR grammars can be extended with the notion of language boxes, and how one can edit those language boxes in the language composition editor *Eco*.

5.1 Introduction

To explain the inner workings of *Eco* and language boxes, the following sections use as a running example a composition of HTML, Python, and SQL, leading to the construction of a flexible system equivalent to 'pre-baked languages' like PHP. In essence, it is shown how a user can take modular languages, compose them, and use the result in *Eco* as shown in Figure 5.1. This section outlines how this example composition is defined and used from the perspective of a 'normal' end-user; the remaining sections are devoted to explaining the techniques which make this use case possible, as well as explaining how important corner cases are dealt with.

[1] Though note there are unambiguous grammars that are not contained within LR(k).

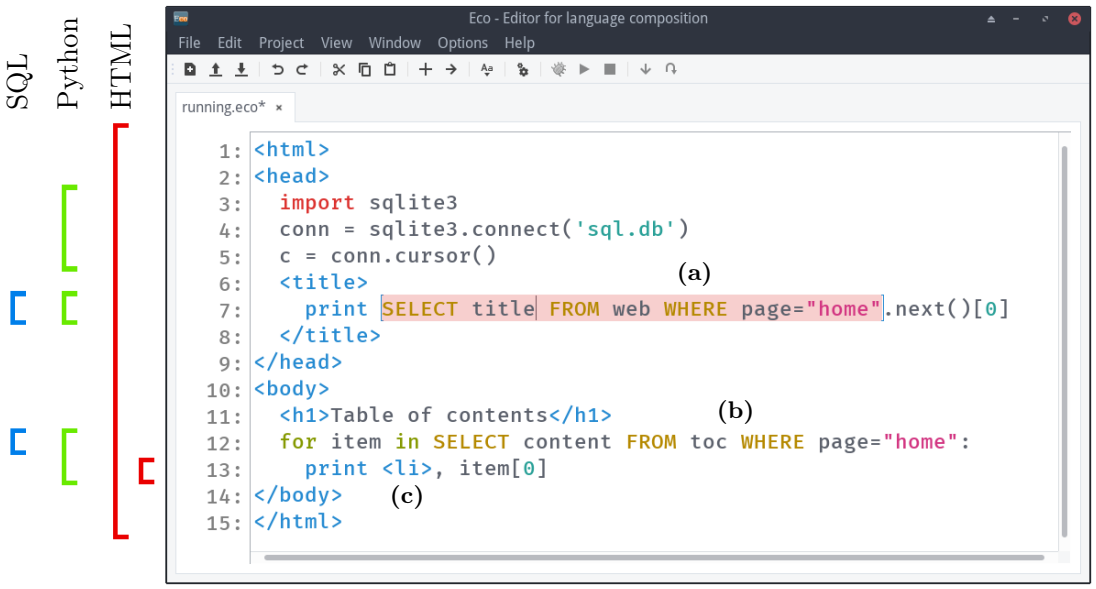


Figure 5.1: *Eco* editing a composed program. An outer HTML document contains several Python language boxes. Some of the Python language boxes themselves contain SQL language boxes. Some specific features are as follows. **(a)** A highlighted (SQL) language box (highlighted because the cursor is in it). **(b)** An unhighlighted (SQL) language box (by default *Eco* only highlights the language box the cursor is in, though users can choose to highlight all boxes). **(c)** An (inner) HTML language box nested inside Python.

When an end-user creates a new file in *Eco*, they are asked to specify which language that file will be written in. Let us assume that they choose the composed language named (unimaginatively) HTML+Python+SQL which composes the modular HTML, Python, and SQL languages within *Eco*. Although users can write whatever code they want in *Eco*, this composed language has the following syntactic constraints: the outer language must be HTML; in the outer HTML language, Python language boxes can be inserted wherever HTML elements are valid (i.e. not inside HTML tags); SQL language boxes can be inserted anywhere a Python statement is valid; and HTML language boxes can be inserted anywhere a Python statement is valid (but one cannot nest Python inside such an inner HTML language box). Each language uses *Eco*'s incremental parser-based editor.

From the user's perspective, their typical workflow for a blank document is to start typing HTML exactly as they would in any other editor: they can add, alter, remove, or copy and paste text without restriction. The HTML is continually parsed by the outer language box's incremental parser and a parse tree constructed and updated appropriately within the language box. Syntax errors are highlighted as the user types with red squiggles. The HTML grammar has been modified to specify where Python+SQL language boxes are syntactically valid by referencing a separate, modular Python grammar. When the user wishes to insert Python code, they press Ctrl + L, which opens a menu of

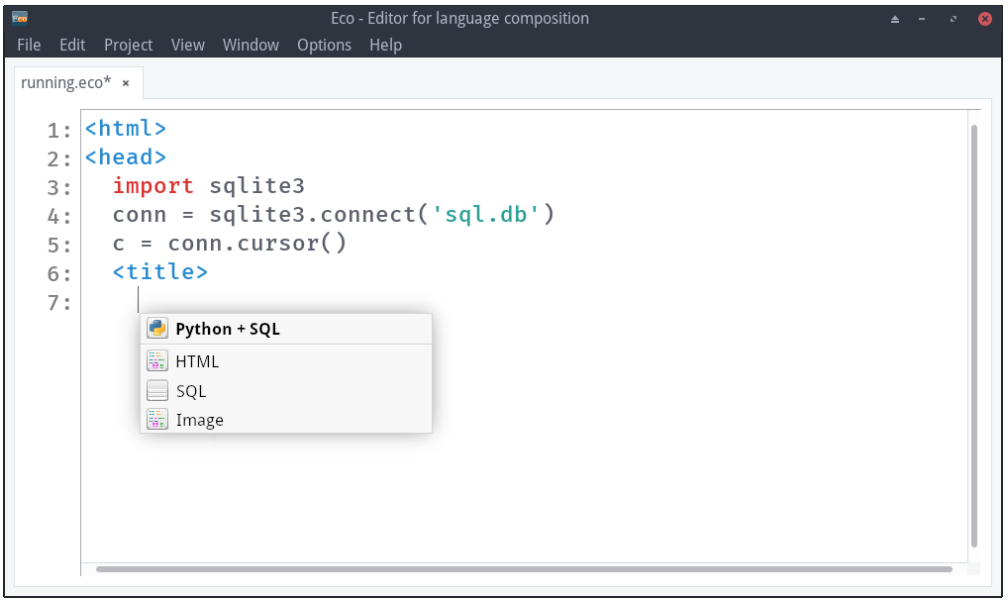


Figure 5.2: Inserting a language box opens up a menu of the languages that *Eco* knows about. Languages which *Eco* knows are valid in the current context are highlighted in bold to help guide the user.

available languages (see Figure 5.2); they then select Python+SQL from the languages listed which inserts a Python language box into the HTML they had been typing. The Python+SQL language box can appear at any point in the text; however, until it is put into a place consistent with the HTML grammar's reference to the Python+SQL grammar, the language box will be highlighted as a syntax error. Note that this does not affect the user's ability to edit the text inside or outside the box, and the editing experience retains the feel of a normal text editor. As Figure 5.3 shows, *Eco* happily tolerates syntactic errors – including language boxes in positions which are syntactically invalid – in multiple places.

Typing inside the Python+SQL language box makes it visibly grow on screen to encompass its contents. Language boxes can be thought of as being similar to the quoting mechanism in traditional text-based approaches which use brackets such as ⟦ ⟧; unlike text-based brackets, language boxes can never conflict with the text contained within them. Users can leave a language box by clicking outside it, using the cursor keys, or pressing Ctrl+Shift+L. Within the parse tree, the language box is represented by a token whose type is Python+SQL and whose value is irrelevant to the incremental parser. As this may suggest, conceptually the top-level language of the file (HTML in this case) is a language box itself. Each language box has its own editor, which in this example means each has an incremental parser.

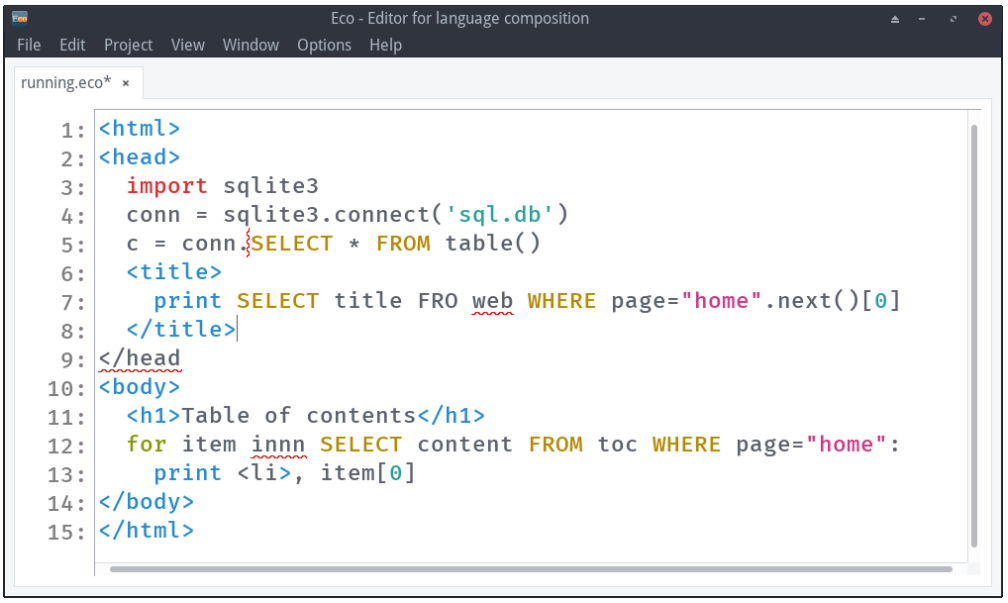


Figure 5.3: Editing a file with multiple syntax errors. Lines 7, 9 and 12 contain syntax errors in the traditional sense, and are indicated with horizontal red squiggles. A different kind of syntax error has occurred on line 5: the SQL language box is invalid in its current position (indicated by a vertical squiggle).

At the end of the editing process, assuming that the user has a file with no syntax errors, they will be left with a parse tree with multiple nested language boxes inside it as in Figure 5.1. In other words, the user will have entered a composed program with no restrictions on where language boxes can be placed; with no requirement to pick a bracketing mechanism which may conflict with nested languages; with no potential for ambiguity; and without sacrificing the ability to edit arbitrary portions of text (even those which happen to span multiple branches of a parse tree, or even those which span different language boxes).

Eco saves files in a custom tree format so that, no matter what program was input by the user, it can be reloaded later. In the case of the HTML+Python+SQL composition, composed programs can be exported to a Python file and then executed. Outer HTML fragments are translated to print statements; SQL language boxes to SQL API calls (with their database connection being set to whatever variable a call to `sqlite3.connect` was assigned to); and inner HTML fragments to strings. All of the syntactically correct programs shown here can thus be run as real programs. Other syntactic compositions, and other execution models of composed programs are possible (see Sections 5.4, 5.5) and there is no requirement for *Eco* compositions to be executable or savable as text.

5.2 Language boxes

Language boxes allow users to embed one language inside another. Language boxes have a type (e.g. HTML), an associated editor (e.g. an extended incremental parser), and a value (e.g. a parse tree). By design, language boxes only consider their own contents ignoring parent and sibling language boxes. It is therefore necessary to define the notion of the CST (Concrete Syntax Tree), which is a language box agnostic way of viewing the user's input. Different language box editors may have different internal tree formats, but each exposes a consistent interface to the CST. Put another way, the CST is a global tree which integrates together the internal trees of individual language boxes.

5.2.1 Language modularity

To make language boxes practical, languages need to be defined modularly, i.e. each language is defined separately and may have several sub-components (e.g. grammar, name binding rules, syntax highlighting). *Eco* allows users to define as many languages as they wish. In order to create a composition, we can alter the grammar of a language L to reference another language M by adding a symbol `<M>` to one or more of L's production rules.

In most cases, however, users will want to avoid hard-coding references to different languages into 'pure' grammars. It is therefore allowed for grammars to be cloned and (during initialisation only) mutated automatically. The most common mutation is to add a new alternative to a grammar. For example, if we have a reference to `python` and `sql` languages, we can create a reference from Python to SQL by executing `python.add_alternative("atom", sql)`. This would be equivalent to altering the Python grammar by adding the alternative "`<SQL>`" to the `atom` rule:

```
atom ::= "NAME"
       | "NUMBER"
       ...
       | atom_loop
       | <SQL>;
```

Another common mutation is to create a subset of a language by changing the start rule of a grammar. For instance, if we want to create a new language that only consists of Python expressions, we can use `python.change_start("expr")`.

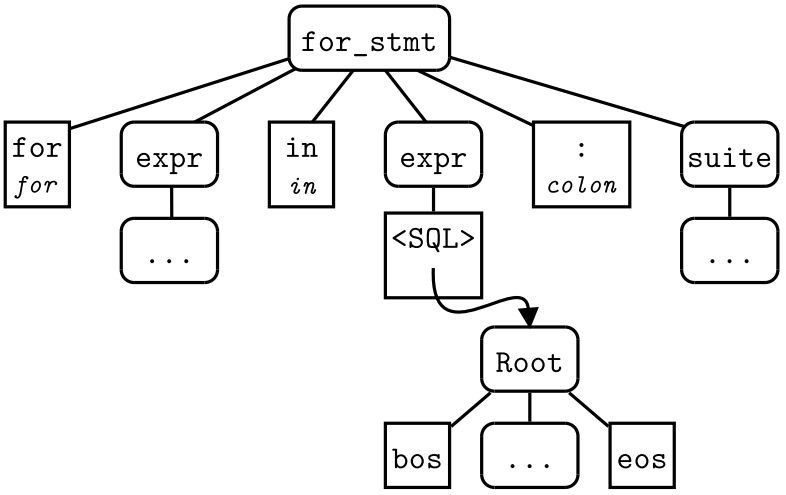


Figure 5.4: An elided example of an SQL language box nested within an outer Python language box. From the perspective of the incremental parser, the tree stops at the SQL token. However, we can clearly see in the above figure that the SQL language box has its own parse tree, which thus forms part of the wider CST.

5.2.2 Language boxes and incremental parsing

Language boxes fit naturally with the incremental parser due to a property of CFGs which is rarely of consequence to batch-orientated parsers: parsers only need to know the type of a token and not its value. In this incremental parser approach, nested language boxes are therefore treated as tokens. When the user inserts an SQL language box into Python code, a new node of type `SQL` is inserted into the parse tree and treated as any other token. From the perspective of the incremental parser for the Python code, the language box's value is irrelevant as is the fact that the language box's value is mutable. Language boxes can appear in any part of the text, though, in this example, an SQL language box is only syntactically valid in places where the Python grammar makes a reference to the SQL grammar. Nested language boxes which use the incremental parser have their own complete parse trees, as can be seen in Figure 5.4.

5.2.3 Impact on rendering

While language boxes do not have any impact on the incremental parser, they do have a big effect on other aspects of *Eco*. One obvious change is that they break the traditional notion that tokens are n characters wide and 1 line high. Language boxes can be arbitrarily wide, arbitrarily high, and don't need to contain text at all. *Eco* cannot simply store text 'flat' in memory and render it using traditional text editing techniques. Instead, it must render the CST onto screen. However, efficiency is a concern. Even a small 19KiB

Java file, for example, leads to a parse tree with almost 19,000 nodes. Rendering large numbers of nodes soon becomes unbearably time-consuming.

To avoid this problem, *Eco* only renders the nodes which are currently visible on screen. *Eco* treats newlines in the user's input specially and uses them to speed up rendering. Similar to Harrison [36], *Eco* maintains a list of all lines in the user's input; whenever the user creates a newline, a new entry is added. Each entry stores a reference to the first CST node in that line and the line's height. Entries are deleted and updated as necessary. Scanning this list allows *Eco* to quickly determine which chunks of the CST need to be rendered, and which do not. Even in this simple implementation, this approach scales to tens of thousands LoC without noticeable lag in rendering.

5.2.4 Cursor behaviour

In a normal editor powered by an incremental parser, cursor behaviour can be implemented as in any other editor and stored as a *(line#, column#)* pair. This approach was initially used for *Eco*, but it has an unacceptable corner-case: nested language boxes create 'dead zones' where it is impossible to place the cursor and to enter further text.

The solution is simple: *Eco*'s cursor is relative to nodes in the CST. In textual languages, the cursor is a pair *(node, offset)* where *node* is a reference to a token and *offset* is a character offset into that token. In normal usage, the arrow keys work as expected. For example, when the cursor is part way through a token, [→] simply increments *offset*; when *offset* reaches the end of a token, [→] sets *node* to the next token in the parse tree and *offset* to 1. [↑]/[↓] is slightly more complex: *Eco* scans from the beginning of the previous / next line, summing up the width of tokens until a match for the current x coordinate is found.

At the end of a nested language box, pressing [→] sets *node* to the next token after the language box while setting *offset* to 1 as described above. This means that if two language boxes end at the same point on screen, *Eco* will seemingly skip over the outer of the two boxes, making it impossible to insert text at that point. If instead the user presses [Ctrl]+[Shift]+[L], the cursor will be set to the current language box token itself instead of the first token after the language box (since language boxes are tokens themselves, this adds no complexity to *Eco*). When the user starts typing, this naturally creates a token in the outer language box. In this way, *Eco* allows the user to edit text at any point in a program, even in seemingly 'dead' zones (see Figure 5.5 for a diagrammatic representation).

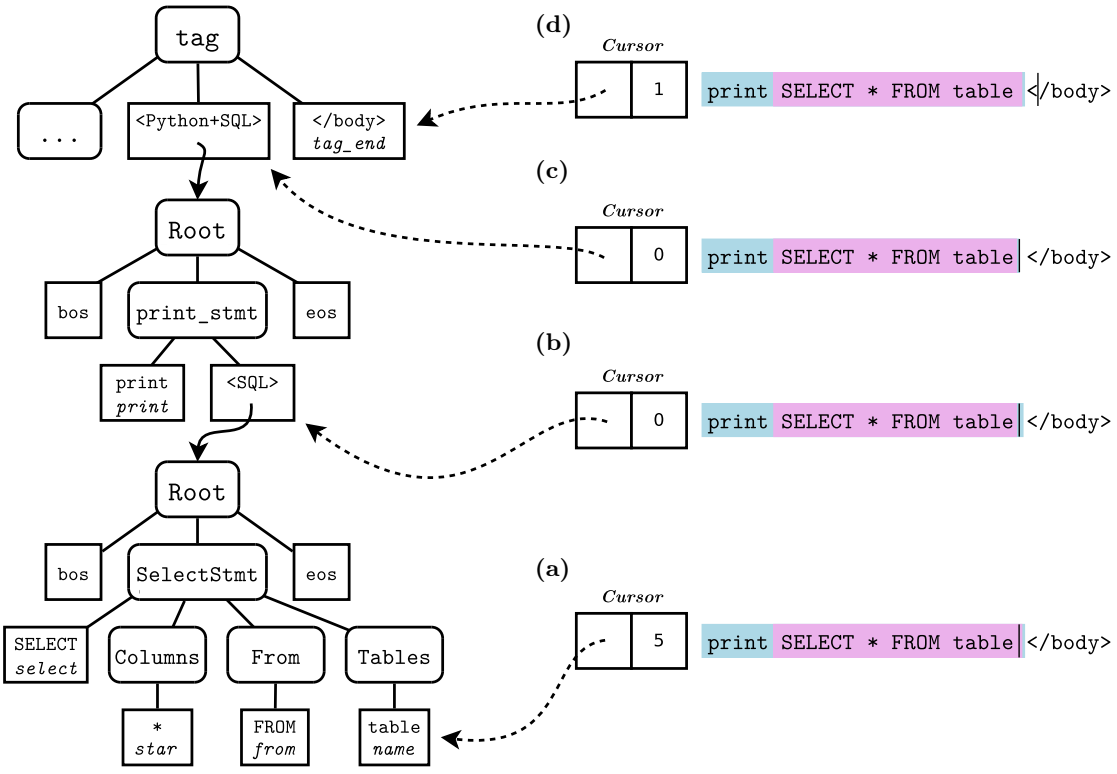


Figure 5.5: *Eco*'s cursor behaviour in a program nesting SQL inside Python inside HTML. The cursor is stored as a (node, offset) pair. **(a)** In normal program editing, the cursor behaves exactly like any other editor. Typing with the cursor at this position will enter text into the SQL language box right after the `table` token. **(b)** After pressing Ctrl+Shift+L, the cursor attaches itself to the current node's language box (`<SQL>`). Typing with the cursor at this position will insert text into the Python+SQL language box between the tokens `<SQL>` and `EOS`. **(c)** After pressing Ctrl+Shift+L again, typing will insert text into the HTML outer language box (after the Python+SQL language box, and before the `</body>` token). **(d)** Assuming the cursor was as in position **(a)** and the user pressed →, the cursor will be moved to this position.

When the cursor is moved to the boundary of a language box, i.e. the cursor is moved to the end or the beginning of a box, the cursor always stays within the language box it started at. Ctrl+Shift+L then either enters or exits the box, depending on the location of the cursor relative to the box.

5.2.5 Copy and paste

Eco allows users to select any arbitrary fragment of a program, copy it, and paste it in elsewhere. Unlike an SDE, *Eco* does not force selections to respect the underlying parse tree in any way. Users can also select whole or partial language boxes, and can select across language boxes. *Eco* currently handles all selections by converting them into 'flat' text and re-parsing them when they are pasted in. This seems like a reasonable backup solution since it is hard to imagine what a user might expect to see when a

partial language box is pasted in. However, some special-cases might be better handled separately: for example, if a user selects an entire language box, it would be reasonable to copy its underlying tree and paste it in without modification.

5.3 Other features

5.3.1 Syntax highlighting

The colouring of syntax is for many programmers one of the most important quality-of-life improvements of programming editors as it significantly improves the readability of their code [69]. Typically, syntax highlighting is implemented either via heuristic pattern matching (i.e. regular expressions) or via parsing. While the former often leads to inaccurate results, the latter can be very time consuming in large programs. Modern IDEs, like Eclipse, have adapted to this problem by parsing small portions of the source code using hand-coded incremental parsing [46] or parse the program in parallel to the user typing, while others, like Atom have opted for a full incremental parser. Since *Eco* is based on a full incremental parser, it doesn't have these problems and thus adding syntax highlighting is simple and straight forward. *Eco* employs syntax highlighting rules similar to [40], which tell the renderer how to colour or highlight certain tokens. Highlighting behaviour can also easily be improved by using parse tree information: for example, in Python this allows us to highlight a function name identifier differently than a variable identifier, even though they both share the same token type. Also, since each language box has its own parser and lexer, the editor naturally highlights code within a language box independently from the outer language, without any modification to the renderer.

5.3.2 Scoping rules

Modern IDEs calculate the available variable names in a source file for code completion, and highlight references to undefined names. *Eco* implements (a subset of) the NBL approach [45] which defines a declarative language for specifying such scoping rules. This runs over the AST created by Section 4.1. References to undefined variables are highlighted with orange squiggles. Users can request code completion on partially completed names by pressing Ctrl+Space. Code completion is semi-intelligent: it uses NBL rules to only show the names visible to a given scope (e.g. variables from different methods do not 'bleed' into each other). There was no need to make changes to the core of *Eco* to make this work. I suspect that other analyses which only require a simple AST will be equally easy to implement.

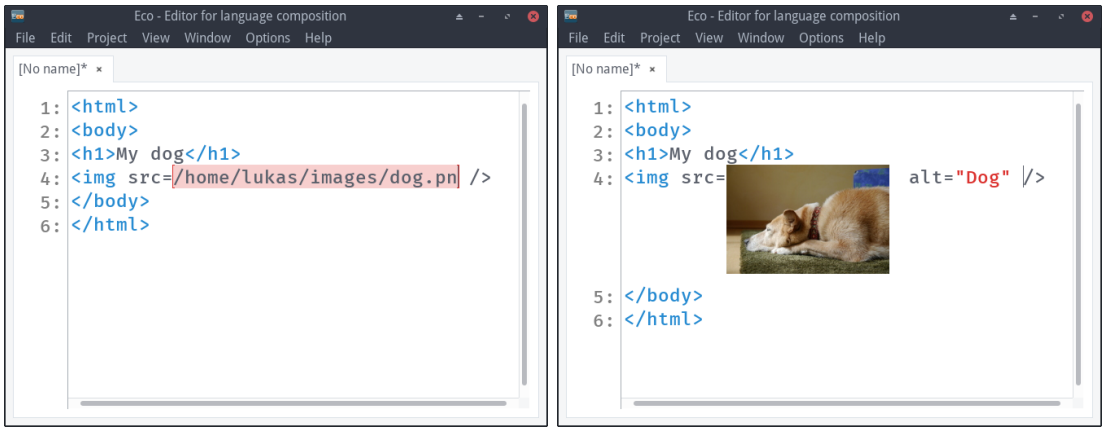


Figure 5.6: An example of a non-textual language in *Eco*. The composition allows images to be embedded into HTML and rendered directly within the editor. Users can toggle between different representations by double clicking the language box.

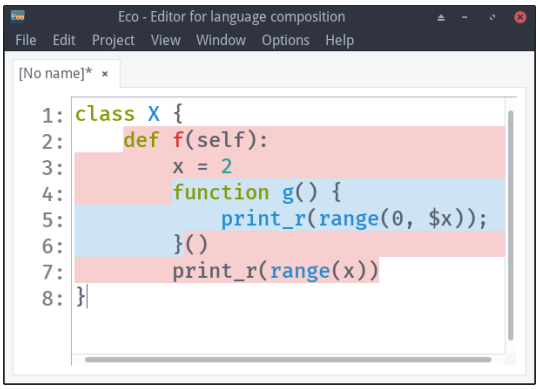
5.3.3 Non-textual languages

Although this chapter's main focus has been on textual languages, language boxes liberate us from only considering textual languages. As a simple example of this, the HTML language we defined earlier can use language boxes of type `Image`. Image language boxes reference a file on disk. When an HTML file is saved out, they are serialised as normal text. However, the actual image can be viewed in *Eco* as shown in Figure 5.6. Users can move between text and image rendering of such language boxes by double-clicking on them. The renderer correctly handles lines of changing heights using the techniques outlined in Section 5.2.3.

As this simple example may suggest, *Eco* is in some senses closer to a syntactically-aware word processor than it is a normal text editor. Although non-textual languages aren't explored in great detail in this thesis, it is easy to imagine appropriate editors for such languages being embedded in *Eco* (e.g. an image editor; or a mathematical formula editor).

5.4 Case study: Unipycation

Eco has been used as part of a case study, Unipycation [7], which investigated the basic interpreter composition of the languages Prolog and Python. The composition allows one-way embeddings, i.e. Prolog can be embedded into Python. *Eco* was used to write a composed program in the form of a *Connect4* game where all GUI elements were written in Python, while Prolog was used for the AI. The program was then exported from *Eco* into a format that the composed Python+Prolog VM can understand. Due to the




```
1  <?php{
2  class X {
3  }
4  compile_py_meth("X", "def f(self):\n \\
5    x = 2\n \\
6    compile_php_func(\"\"\"function g() {\n
7      print_r(range(0, \$x));\n
8    }\"\"\")()\n \\
9    print_r(range(x))");
10 }?>
```

Figure 5.7: An example of a PyHyp composition with nested language boxes (on the left) and its exported, textual representation (on the right). The example shows that writing compositions by hand without the help of language boxes is difficult as embedded languages and even newlines (for languages that don't support multiline strings, such as Java) need to be escaped manually.

simplicity of the composition, this only required wrapping the content of the language box inside a string and escaping strings within the box. Though basic, this case study already showed *Eco*'s usefulness as writing even such basic compositions by hand, requiring the manual escaping of strings, turned out to be rather cumbersome.

5.5 Case study: PyHyp

A second case study where *Eco* was utilised to write compositions is PyHyp [6]. Using more fine grained compositions, PyHyp allows multiple and even nested embeddings between the two languages PHP and Python. Python code can be embedded into PHP code at any point where a PHP statement would be valid, e.g. class/function declarations, if-blocks, for/while-loops, method calls, expressions, etc. Additionally, it allows a subset of Python, expressions, to be embedded into PHP wherever PHP expressions are valid. Separating compositions this way, i.e. embedding Python expressions separately from normal Python language boxes, allows us to treat those boxes differently when exporting the composed program according to the requirements imposed by the composed VM.

PyHyp further extends compositions to being able to nest languages deeper than the two levels seen in the first case study. For instance it is possible to embed a Python language box into PHP and then embed a PHP language box into the Python language box. Further embeddings are possible. With multiple embeddings, especially nested ones, writing PyHyp programs using a traditional text editor quickly becomes a tedious and error prone task (see Figure 5.7). *Eco* also helps dealing with cross-language line numbers; when a runtime error occurs within the code of a language box, its line number can be accurately mapped back to the right location in the editor. This further emphasises the importance of an editor like *Eco* when dealing with composed programs.

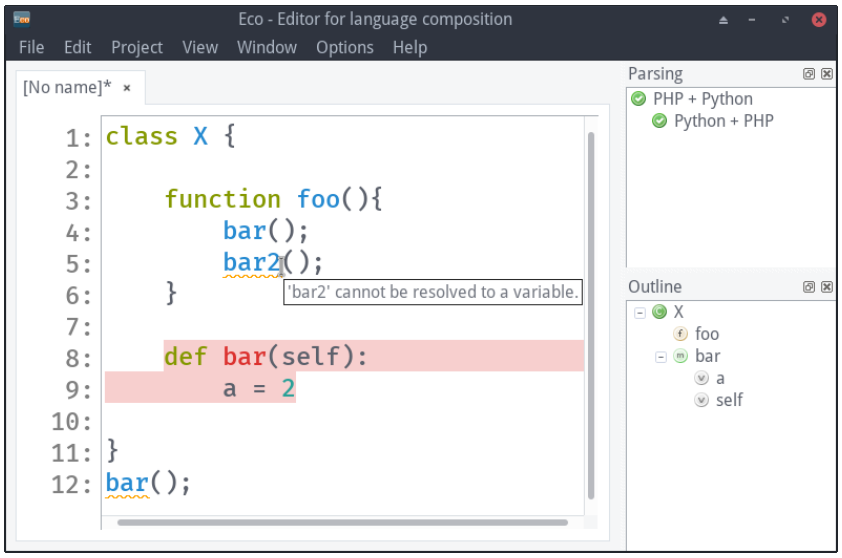
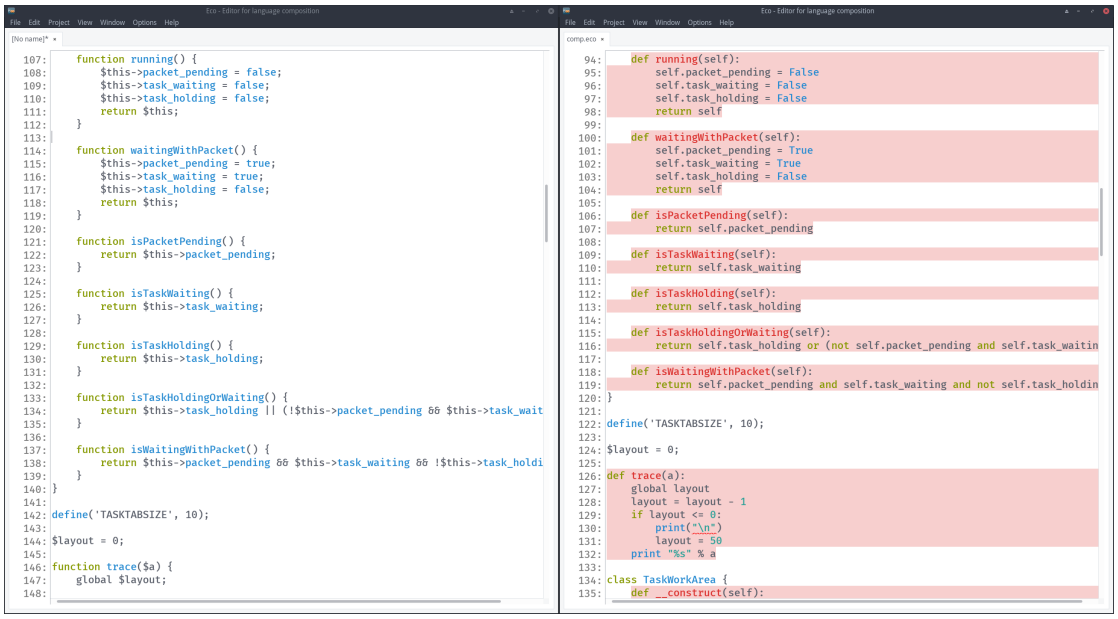


Figure 5.8: An example showing name binding across languages. A Python method `bar` has been embedded into a PHP class. The outline (bottom right) shows that the editor considers the Python method as part of the PHP class. Referencing `bar` within another PHP function resolves correctly to the Python method (line 4), while referencing the method outside of the class (line 12) or referencing undefined methods (line 5) generates warnings as one would expect.

5.5.1 Cross-language scoping

Eco also supports basic cross-language scoping. In PyHyp, both Python and PHP define their own name binding rules as described in Section 5.3.2. In order to support scoping across language boxes, we can simply merge together the results of the name binding analysis of each language, with minor adjustments. For example, when merging a Python method into a PHP class, we have to adjust the method's path (i.e. within the language box the method is at the top level, but when merged into PHP it needs to be at the class level). For example, if we add the Python method `bar` into a PHP class `X`, where `X` has the path `class:X/` and `bar` has the path `func:bar/`, then the method is merged by adding it to PHP's name binding results, while changing its path to `func:X/bar`. Furthermore, languages may use different names to define name binding types, e.g. in PHP a function definition may be called `func` while in Python it is called `method`. This means, that when merging a Python method `bar` in a PHP class `X`, its new path would be `method:X/bar`, instead of `func`. In such cases it is thus necessary to extend the name binding rules of a language with additional references. For example, in PyHyp the name binding rules for PHP were extended so that a `FunctionCall` references both `func` and `method`. Visibility is always defined by the outer language of the composition, e.g. in order to decide if a PHP function is visible inside a Python function, Python follows the scoping rules of the outer PHP language. Figure 5.8 shows an example of cross-language scoping in *Eco*. Appendix D shows a more detailed description of cross-language scoping including exemplary name binding rules.



```
function running() {
    $this->packet_pending = false;
    $this->task_waiting = false;
    $this->task_holding = false;
    return $this;
}

function waitingWithPacket() {
    $this->packet_pending = true;
    $this->task_waiting = true;
    $this->task_holding = false;
    return $this;
}

function isPacketPending() {
    return $this->packet_pending;
}

function isTaskWaiting() {
    return $this->task_waiting;
}

function isTaskHolding() {
    return $this->task_holding;
}

function isTaskHoldingOrWaiting() {
    return $this->task_holding || (!$this->packet_pending && $this->task_wait
}

function isWaitingWithPacket() {
    return $this->packet_pending && $this->task_waiting && !$this->task_holdi
}
}

define('TASKTABSIZE', 10);

$layout = 0;

function trace($a) {
    global $layout;
```

```
def running(self):
    self.packet_pending = False
    self.task_waiting = False
    self.task_holding = False
    return self

def waitingWithPacket(self):
    self.packet_pending = True
    self.task_waiting = True
    self.task_holding = False
    return self

def isPacketPending(self):
    return self.packet_pending

def isTaskWaiting(self):
    return self.task_waiting

def isTaskHolding(self):
    return self.task_holding

def isTaskHoldingOrWaiting(self):
    return self.task_holding or (not self.packet_pending and self.task_waitin

def isWaitingWithPacket(self):
    return self.packet_pending and self.task_waiting and not self.task_holdin
}

define('TASKTABSIZE', 10);

$layout = 0;

def trace(a):
    global layout
    layout = layout - 1
    if layout <= 0:
        print("\n")
        layout = 50
    print "%s" % a

class TaskWorkArea {
    def __construct(self):
```

Figure 5.9: The Richards benchmark used in PyHyp. All benchmarks were written in three versions: PHP, Python, and PHP+Python. The figure shows the mono PHP version on the left, and the composed PHP+Python version on the right.

5.5.2 Benchmark migration

For the evaluation of PyHyp, the case study tested the performance of compositions against their single-language equivalents. For this, three versions needed to be created for each benchmark: one in PHP, one in Python, and one in PHP composed with Python. All benchmarks started out as PHP programs. Each program was then converted into Python, by manually replacing one PHP method at a time with a Python equivalent, thus slowly migrating the PHP program into Python. Figure 5.9 shows an example.

5.5.3 A SquirrelMail plug-in

SquirrelMail is a venerable PHP web mail client. *PyHyp* and *Eco* were used to add a SquirrelMail plug-in that uses the Python SymPy library. This is intended to show that *PyHyp* can be used to add Python modules to relatively large existing systems. In essence, the plug-in recognises mathematical formulae between triple backticks, and uses SymPy to render them in traditional mathematical notation. Formulae in incoming emails are automatically rendered; users sending emails with such formulae can preview the rendering before sending. Figure 5.10 shows the plug-in in use, and the core parts of the code within Eco.

The `sympy_changebody_do` function is called by SquirrelMail's `message_body` hook (which is also called upon viewing a message), receiving the content of the email as an argument.

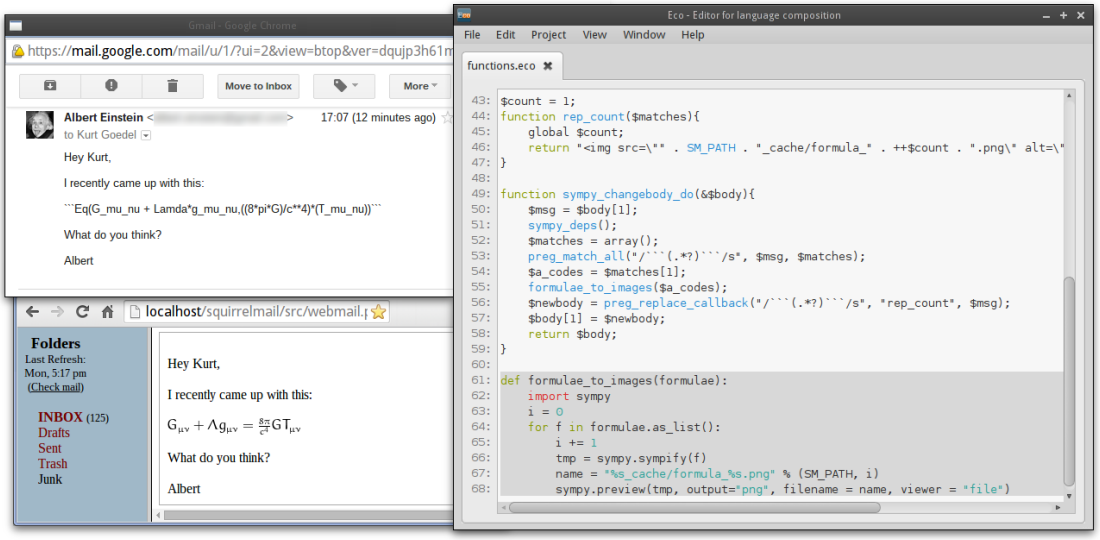


Figure 5.10: Example mails sent with the extended version of SquirrelMail. The PHP mail client was extended such that it can visualise mathematical formulae using the SymPy Python library. A portion of the plug-in code is shown in the right.

A regular expression finds all occurrences of formulae between backticks (line 53) and passes them to the Python `formulae_to_images` function. This then uses SymPy to convert the formulae to images (numbered by their offset in the array/list) into the directory pointed to by the PHP constant `SM_PATH` (lines 61–68), and uses the URL of the image in-place of the textual formula (line 56).

5.6 Testing

During *Eco*'s development testing has been an integral part of finding and fixing bugs in all of its components. Many of *Eco*'s bugs were found using fuzz testing, leading to a continuously growing test suite. Two notable examples are the bugs found in Wagner's error recovery algorithm which are described in Sections 3.3.5 and 3.5.3. At the time of writing, *Eco*'s test suite consists of 302 tests, of which 150 specifically test the editor with the remaining testing dependencies such as the parser and lexer.

5.6.1 Fuzz testing

Testing text editors can be difficult, especially when the user base is small. Even if the editor is used throughout its development (e.g. all of *Eco*'s grammars were written within *Eco* itself), this is insufficient to iron out all bugs. A solution to this problem is *fuzz testing* [56], a form of testing where inputs are randomly generated or mutated. *Eco*'s test suite includes a framework which takes as input a grammar and a program for that

```
1  from random import shuffle
2  def fuzzy_deletion_D():
3    load_file(input)
4    log("load_file({})".format(input))
5
6    for line in shuffle(getlines(input)):
7      cols = range(len(line))
8      for col in shuffle(cols)[0:5]: # restrict to 5 edits per line
9        move_cursor(line, col)
10       log("move_cursor({}, {})".format(line, col))
11       key_delete()
12       log("key_delete()")
```

Listing 5.1: Example of a fuzzy tests that loads a program (line 3), moves the cursor to a random line and column in the input (lines 6–9) and deletes a character from that location (line 11). As its name suggests, `shuffle` returns a new list where all elements have been shuffled. Every action is saved to a log, so that if a runtime error occurs, the test can be reproduced. *Eco*'s test suite contains multiple fuzz tests, some of which only insert or delete characters, while others do a combination of different actions including undo and redo.

grammar and then runs a variation of fuzzy tests. These tests simulate different random actions, such as insertion or deletion of characters, moving of the cursor, or undoing or redoing changes, and applying them to the current input (see Listing 5.1 for an example). Each time an action is applied, the input is incrementally re-lexed and re-parsed. If an action breaks the editor, i.e. any form of runtime error occurs, then the test fails and stores a log of all actions used. This log can then be used to reproduce the error and create a stand-alone test to be added to the test suite (an example of such a log can be seen in Listing 5.2).

5.6.2 Minimising test logs

Each time a fuzz test discovers a bug, a log is saved which can be used to reproduce the test and add it to *Eco*'s test suite. Unfortunately, these logs can get very long, using thousands of actions before the actual error occurs, many of which are irrelevant to the bug. This not only unnecessarily increases the overall runtime of the test suite when the test is added to it, but more importantly it makes it harder to find the cause of the error and fix the bug. In most cases the bug can be reproduced with only a handful of actions.

We can reduce the log to those actions by removing any action that doesn't change the outcome of the test generated from that log. However, minimising the log by hand is a time consuming and tedious process as each time an action is removed the test needs to be rerun to make sure its outcome is still the same. Therefore, *Eco*'s testing framework includes a *minimiser*, which heavily reduces the size of a log by removing all actions that are not needed to reproduce the bug. Although its functionality is very basic, the

```
load_file("deltablue.py")
move_cursor(5, 10)
key_delete()
move_cursor(5, 4)
key_delete()
move_cursor(5, 8)
key_delete()
move_cursor(5, 7)
key_delete()
move_cursor(5, 2)
key_delete()
move_cursor(22, 2)
key_delete()
move_cursor(22, 6)
key_delete()
move_cursor(22, 5)
key_delete()
move_cursor(22, 9)
key_delete()
move_cursor(22, 1)
key_delete()
move_cursor(7, 18)
key_delete()
move_cursor(7, 16)
key_delete()
move_cursor(7, 5)
key_delete()
move_cursor(7, 12)
key_delete()
move_cursor(7, 2)
key_delete()
move_cursor(14, 8)
key_delete()
move_cursor(14, 16)
key_delete()
move_cursor(14, 3)
key_delete()
move_cursor(14, 12)
key_delete()
move_cursor(14, 6)
key_delete()
...
```

Listing 5.2: Example log file of a fuzzy test. The log can simply be copied into *Eco*'s test suite to create a test from it. However, since logs can become very large, it is often necessary to reduce their size to decrease their runtime and help with debugging.

minimiser can reduce a log containing thousands of actions to a few dozen. Often, those minimised logs can be reduced further by hand as some actions are not caught by the basic minimiser. For example, sometimes an action, though unnecessary to reproduce the bug, changes the program in a way that causes later actions to trigger the bug. Thus, removing an action that deletes a newline or a character requires adjusting the line numbers and columns of all following actions. However, this increases the complexity of the minimiser and in most cases the basic minimising functions are sufficient to generate tests small enough for debugging.

Minimising a test log is simple: we start at the top and remove one action at a time and rerun the test to check if the minimisation still produces the bug. If the removal of an action changes the error message or avoids the bug all together, the log is reverted to its last 'failing' version. Actions are removed in pairs since one action moves to cursor to a random position and the other deletes or inserts a character. To improve the performance of the minimiser there is an additional mode which groups together actions by the line number they occur in and removes all of them together (e.g. for the log in Listing 5.2 this would remove 10 actions at a time). This *group* mode is run first to get rid of the major bulk of unneeded actions, and is then followed by the more fine-grained *pair* mode described earlier. I also experimented with a binary chop mode which tries to remove the log by half each time. However, while faster in some cases, most tests have their salient

```
1  def minimise_D(log: String, mode: int, error: Exception):
2    changed = True
3    while changed:
4      line = 1 # skip over load_file
5      changed = False
6      while line < len(log):
7        temp = log[:] # create a copy of log
8        del log[line:line+mode] # remove actions
9        try:
10         run(log)
11       except Exception as e:
12         if e == error: # exception remains
13           changed = True
14           continue
15       log = temp
16       line += mode
17   return log
```

Listing 5.3: A function to reduce a log generated from a fuzzy test. The minimising function takes as input a log, a reduction mode (in form of the amount of lines to be removed each iteration), and an error. It then attempts to remove actions from the log according to `mode` (line 8) until no more actions can be removed. After each removal, the test log is rerun to make sure that it still reproduces the same bug (line 9-14). If it doesn't the last removal is reverted (line 15). Otherwise, we continue. Once no more actions can be removed (i.e. no more changes were made to the log), the minimised log is returned (line 17).

parts distributed throughout the log, and are thus not amenable to such a mode. Each mode runs until no more actions can be removed at which point the next mode takes over. Finally, when no more actions can be removed, the log has been minimised and is stored to disk. Listing 5.3 shows the `minimise` function used for both modes.

Chapter 6

Lexing language boxes inside comments and strings

The introduction of language boxes introduces a new editing problem into the editor: what should happen if we comment out a line containing a language box or insert a language box into a string? A naive solution would be to either ignore the language box or flatten it, but neither solution is desirable, as they lead to errors or loss of information. This chapter first explains the nature of the problem and then introduces *multinodes* which allow language boxes to be integrated into other tokens while retaining their structure and identity.

6.1 The problem

To illustrate the problem, let's assume, as our running example, the following scenario: a user writes a program in which they use a language box to embed an SQL statement into some arbitrary language's expression:

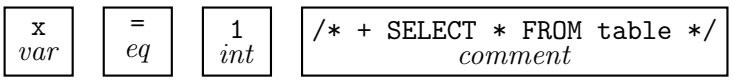
```
x = 1 +  SELECT count(*) FROM table
```

When running the program the user spots unwanted behaviour and suspects that the SQL statement is at fault. In order to debug the program, they comment out the SQL statement, like so:

```
x = 1 /* +  SELECT count(*) FROM table  */
```

This scenario makes it obvious why we need the ability to mix language boxes with comments, however it is not clear how this can be achieved with a traditional lexer.

The obvious solution is to have the lexer ignore the language box boundaries and just process its content like any other code. This results in the flattening of the language box and integration of its content into the comment token, producing the following tokenisation of the program:



While this solves the problem, as the language box has now been successfully commented out and will be ignored by the parser, it creates another problem: flattening the language box has destroyed its parse tree. When the comment is removed again, the editor is unable to recover its original structure, as the previous contents of the language box will now be lexed using the outer language's lexer. Additionally, any meta-data stored within its nodes will be lost as well.

6.2 Outlines of a solution

Ideally, we want to allow the user to comment out language boxes or insert them into strings without destroying them, while hiding them from the parser at the same time. Their content, however, must remain visible to the user and the source code renderer to allow further editing.

The solution to this is to introduce another node type, the *multinode*. This is almost identical in nature to language boxes: the outer parser perceives such nodes as tokens, therefore only needing to check their type, but the renderer is able to access their children. A commented out language box is thus housed under a multinode 'comment': the language box retains all of its settings, but the outer parser sees only a node with type 'comment'. Unlike language boxes, whose content is completely separated from the outer language's lexer and parser, multinodes can be re-lexed by the lexer and their content can be moved outside of the node or combined with other multinodes. On the other hand, multinodes cannot, and don't need to be nested like language boxes, since from a parsing perspective, they still represent a single token which can't be nested. Figure 6.1 shows how our running example would be represented inside the parse tree using multinodes.

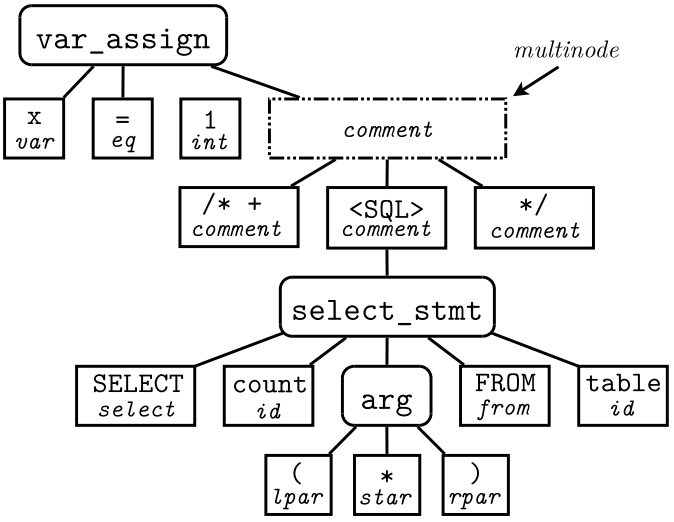


Figure 6.1: An example of an elided parse tree where a language box has been included inside a comment token using a multinode. Since the multinode appears as a token and is of type `comment` it is being ignored by the parser. However, both lexer and renderer can access its children for future processing and displaying. More importantly the language box's contents remain intact.

6.3 Incrementally lexing multinodes

The support for multinodes requires no changes to the parser. However, we need to modify both the lexer (i.e. the tokeniser), which creates tokens from parse tree nodes, and the incremental lexer, which merges those tokens back into the parse tree. We first need to extend the lexer, allowing it to process language boxes when they appear within tokens such as strings and comments, which is described in Section 6.3.1. We then modify the incremental lexer to create multinodes during the merging phase, when appropriate, while continuing to reuse as many nodes as possible, which is described in Sections 6.3.2, 6.3.3, and 6.3.4.

6.3.1 Matching language boxes in the lexer

We can easily extend a lexer with support for language boxes: the idea is to treat a language box like a special character. While the language box cannot be directly referenced within lexing rules, it is caught by wildcards, such as the match-all pattern (e.g. '`.*`'), and negated character ranges (e.g. '`[^a]+`'). This allows the lexer to match language boxes within rules such as comments (e.g. '`//[^\n]*`' or '`/\*.*?\*/`') or strings (e.g. '`\"[^\"\n]*\"`'). Listing 6.1 shows a small fragment of a lexer with extensions that allow it to match language boxes.

When the lexer matches text that contains a language box, it returns a token whose value is a list of strings and language boxes instead of a single string (from here on we call

```
1  def match_one_D(node: Node, pos: int, pattern: RegexPattern) -> bool:
2      if pattern == ".":
3          # Wild card always matches
4          return True
5
6      if type(node) is Languagebox:
7          return False
8
9      char: String = getchar(node, pos) # get character at pos in node's value
10     if pattern == char:
11         return True
12     return False
13
14 def match_range_D(node: Node, pos: int, pattern: RegexPattern) -> bool:
15     if type(node) is Languagebox:
16         if pattern.neg:
17             # Negated char ranges can never exclude a language box
18             return True
19         return False
20
21     char: String = getchar(node, pos)
22     if (pattern.neg and ord(char) not in pattern) \
23                 or (not pattern.neg and ord(char) in pattern):
24         return True
25     return False
```

Listing 6.1: Excerpt of a lexer that operates on parse tree nodes. Support for language boxes has been added by extending the methods responsible for matching characters. The method `match_one` matches exactly one character, while `match_range` matches a normal or negated range of characters (e.g. '`[a-z]`' or '`[^"]`'). Language boxes automatically match the wildcard '`.`' (lines 2–4) but can't be matched by any other character (lines 6–7). A character range can only match a language box if the character range is negated (line 16–18) and cannot be matched otherwise (line 19).

such a token a *multitoken*). For instance, lexing the running example would result in the following tokens (where `lbox` represents the language box within the comment):

```
('x', id), ('=', eq), ('1', int), (['/* +', lbox, '*/'], comment)
```

When the incremental lexer merges these tokens back into the parse tree, tokens that contain multiple items are converted into multinodes, leading to the parse tree seen in Figure 6.1.

6.3.2 Merging multinodes into the parse tree

Naively implemented, creating multinodes and merging them back into the parse tree is simple: first we remove all re-lexed nodes from the parse tree, and then insert the new tokens back into the tree. If, as a result of incremental lexing, a token X is replaced by

a list of tokens (i.e. a token containing a language box), then we create a multinode of the same type as X and insert the list of tokens into that node. Thus, from the parser's perspective, there is still a single token as before; but from the user's (and the renderer's) perspective there is now a list of tokens at that point. As described in Section 2.5, this naive way of merging nodes back into the parse tree unnecessarily destroys and recreates many nodes (including multinodes), thus increasing memory usage. The following sections describe how we can extend the optimised merging algorithm from 2.5.7 to intelligently merge multinodes and reuse old nodes wherever possible.

The extended algorithm is explained in Sections 6.3.3 and 6.3.4. The addition of multinodes adds another level of complexity to the merging algorithm and thus may make some of its details hard to understand. The aim of the following sections is thus to give an overview of the algorithm, while Section 6.4 contains several examples that may help the reader understand its finer details.

6.3.3 Iterating multitokens and multinodes

The `merge_back` algorithm from Section 2.5.7 receives as input two lists, one containing newly generated tokens and another containing the nodes in the parse tree that were processed during re-lexing. Previously, the algorithm could simply iterate over the two lists and merge new tokens from the first with old tokens from the second list. However, with the introduction of multinodes, both lists can now contain a new element type: generated tokens may include multitokens, while processed nodes may contain multinodes. To deal with them appropriately, we create two generator functions, which iterate over the lists and deal with the new types by iterating over and returning their children instead of returning the multitoken/multinode itself (see Listing 6.2).

Processing the children of multitokens and multinodes separately makes it easier to reuse existing multinodes as well as match generated tokens with existing ones from the previous version of the parse tree. For example, when two multinodes are merged into one, the first multinode can be reused and tokens from the other can simply be moved over to the first. Similarly, when splitting up a multinode into two, we can reuse the multinode for the first part of the split, and for the second, create a new multinode and simply move over the remaining tokens from the first.

6.3.4 A new merge method

The new merge algorithm is shown in Listing 6.3. It extends the merge algorithm from Section 2.5.7, allowing it to merge multitokens back into the parse tree by creating and

```
1  def iter_gen_D(tokens: List[Token]) -> Token:
2    for t in tokens:
3      if t is a multitoken:
4        for x in t.value:
5          yield (x, t.type)
6      else:
7        yield t
8    while True:
9      yield None
10
11 def iter_read_D(nodes: List[Node]) -> Node:
12   for n in nodes:
13     if n is a multinode:
14       # since we are removing elements from the list in
15       # merge_back we need to create a copy to not skip anything
16       for x in list(n.children):
17         yield x
18     else:
19       yield n
20   while True:
21     yield None
```

Listing 6.2: Custom iterators for generated tokens and processed nodes that handle multitokens and multinodes by iterating over their children and returning them one at a time (Lines 3-5 and 13-17). The remaining code is a simple iterator functionally similar to the one used in the old `merge_back` function. Once all elements of a list have been processed the iterator continues to return `None`. This avoids having to catch a `StopException` within the `merge_back` function, which succeeds as soon as both lists return None.

removing multinodes where necessary. The new functionality can be summarised as follows: when tokens that are part of a multitoken are being merged, the algorithm either creates a new multinode or reuses a previous one if it exists. Similar to the original algorithm, the new algorithm also tries, whenever possible, to merge tokens by overwriting existing nodes in the parse tree. Additionally, the new algorithm then also moves those overwritten nodes out of or into multinodes as appropriate. Once all children of a multitoken have been merged, the algorithm continues to merge tokens as normal. The remainder of this section explains the algorithm in more detail.

The new merge algorithm starts by using the new generator functions from Listing 6.2 to create iterators for the generated tokens and processed nodes (lines 2–3). When the algorithm encounters the first child of a multitoken (line 14), this and all following tokens are merged into a multinode `current_mt`. If a multinode already exists at that location, which can be determined from the processed node `old`, it can be reused (lines 51–53). Otherwise a new one is created (line 55). We must keep a list of reused multinodes, so that they can only be reused once, since splitting a multinode into two would otherwise result in the first multitoken reusing the old multinode, and the second multitoken then attempting to reuse it again. Once all elements of a multitoken have been merged, the

```
def merge_back_D(gen: List[Token], pro: List[Node]):
  generated: Iter[Token] = iter_gen(gen)
  processed: Iter[Node] = iter_read(pro)
  old: Node = processed.next()
  new: Token = generated.next()
  last: Node = old.previous_token()
  newlen = oldlen = 0
  reused: Set[Node] = set()

  while old or new:
    if new is first token after multitoken:
      last = current_mt
      current_mt = None
    if new is first child of multitoken:
      current_mt = mk_multi(old, reused)
      insert_after(last, current_mt)

    if oldlen >= newlen + len(new): # Insert
      insert_after(last, new, current_mt)
      last = new
      newlen += len(new)
      new = generated.next()
    elif oldlen + len(old) <= newlen: # Remove
      remove(old)
      oldlen += len(old)
      old = processed.next()
    else: # Overwrite
      old.update(new)
      if old or new are child of multi and old.parent != current_mt:
        remove(old)
        insert_after(last, old, current_mt)
      last = old
      oldlen += len(old)
      newlen += len(new)
      new = generated.next()
      old = processed.next()

def insert_after_D(last: Node, new: Token, mt=None):
  if new is first child of multitoken:
    mt.insert(0, Node(new))
  else:
    last.insert_after(Node(new))

def remove_D(node: Node):
  parent = node.parent
  parent.remove(node)
  if parent is empty multinode:
    remove(parent)

def mk_multi_D(old: Node, reused: Set[Node]) -> multinode:
  if old.parent is a multinode and old.parent not in reused:
    reused.add(old.parent)
    return old.parent
  else:
    return new multinode
```

Listing 6.3: Extension of the `merge_back` algorithm from Listing 2.5, adding support for multinodes and multitokens. For simplicity we assume that `len(None)` returns 0.

algorithm returns to merging tokens as per usual by setting `current_mt` to `None` (lines 11–13). If two multitokens appear immediately next to each other, lines 11–13 first finalise the first multinode and assign it to `last`, so that lines 14–16 can then insert a new multinode immediately afterwards.

The insertion (lines 18–22) and deletion (23–26) parts of the old merge algorithm remain largely the same. The only change is the function `insert_after`, which has been altered so that the first child of a multitoken is inserted at the beginning of the multinode, since inserting it after `last` would otherwise insert it outside of the multinode.

The overwrite part requires some more changes, since it is responsible for moving overwritten nodes in and out of multinodes if necessary (lines 29–31). For example, if part of a multitoken overwrites a normal node, that node needs to be moved inside the multinode. If a normal token overwrites a multinode child, then that child needs to be moved outside of the multinode. Also, if part of a multitoken overwrites a multinode child, but the latter's parent is not the same as the current multinode, it needs to be moved from its old multinode into the new one. In all other cases, nodes are overwritten as per usual without the need to move them (line 28). Anytime a node is moved out of a multinode, whether it's removed entirely or just moved somewhere else, and the multinode becomes empty, it needs to be removed from the parse tree (lines 47–48).

6.4 Examples

This section shows some practical examples of scenarios each highlighting different parts of the algorithm from Listing 6.3. To help the understanding of the algorithm all explanations reference the relevant parts of the algorithm via line numbers. For the sake of brevity all examples have been stripped of whitespace and most nonterminals. Language boxes are represented as single tokens (as they appear to the lexer) eliding their underlying parse trees. While the examples shown here suffice to show all facets of the algorithm, Appendix E includes some additional examples that show how the algorithm handles some other scenarios.

6.4.1 Creating multinodes

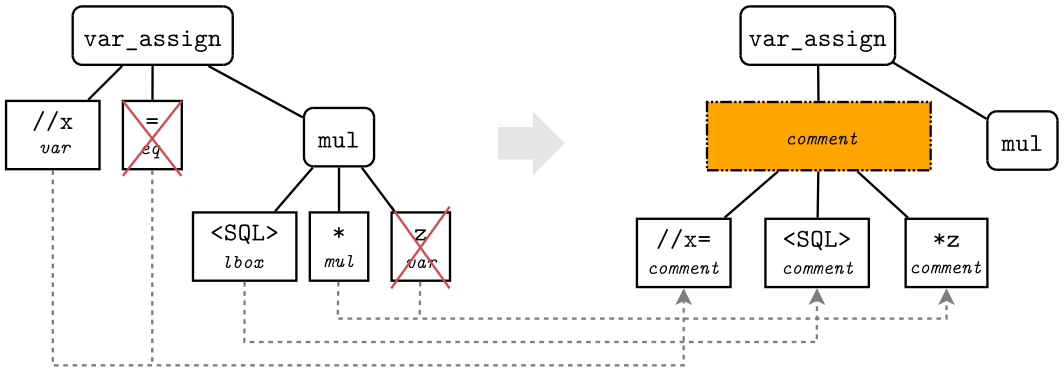
This example shows the creation of a multinode after the user commented out a line containing a language box and uses the *Overwrite* and *Remove* branches of the algorithm. Consider the following program:

```
//x = SELECT x FROM y * z
```

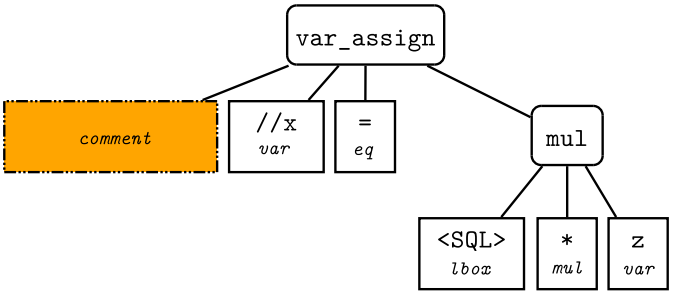
After re-lexing, the lexer returns the following lists of generated tokens and processed nodes which are passed over to the incremental lexer's merging algorithm:

generated tokens	**processed nodes**
`(['//x=', lbox, '*z'], comment)`	`//x` `=` `<SQL>` `*` `z`

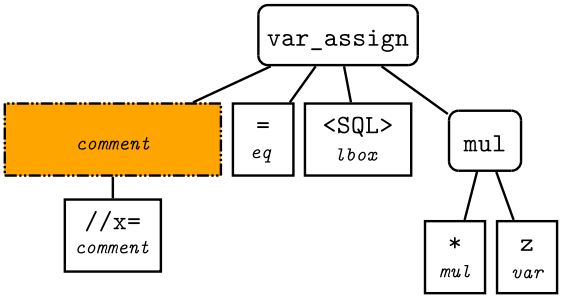
The lexer has combined all nodes into a single multitoken of type *comment*. This means the algorithm needs to create a new multinode and move all processed nodes from the parse tree into this node. At the same time it has to combine the nodes `//x` and `=` as well as `*` and `z` into a single node, update their types and values and remove the left-over nodes. The following figure shows the transition from the old parse tree to the new one (crossed out nodes mean they have been deleted from the parse tree; the arrows show nodes moving from their old location to the new one within the multinode; newly created nodes are coloured orange, merely updated nodes remain white):



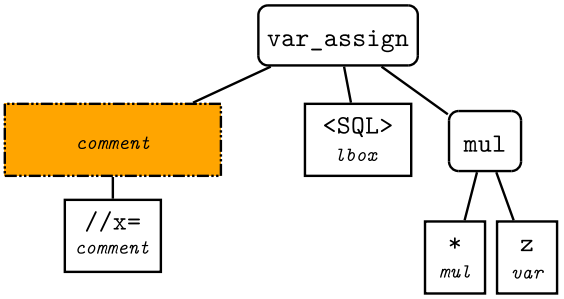
The following explains the merging process step-by-step, referencing lines within the merge algorithm from Listing 6.3 and showing a representation of the parse tree after each step. After initialising the generators, `new` is set to the first generated token '`//x=`' while `old` is set to node `//x`. Since `new` is the first child of a multitoken, we create a new multinode, assign it to `current_mt`, and insert it into the parse tree (lines 12–13):



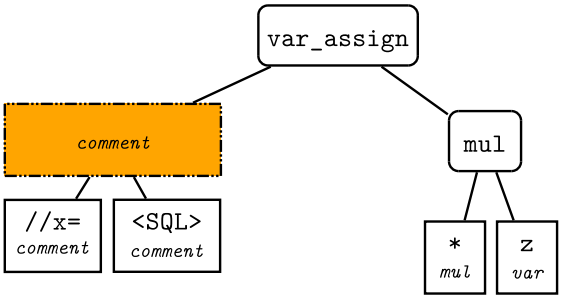
Since both values `oldlen` and `newlen` are 0 we end up in the *Overwrite* branch of the algorithm. We first overwrite the value of the node `//x` with '`//x=`' and set its type to *comment* (line 28). Since `new` is a multitoken child and `old`'s parent is not `current_mt`, we remove `//x=` from the parse tree and insert it into `current_mt` (line 23–32):



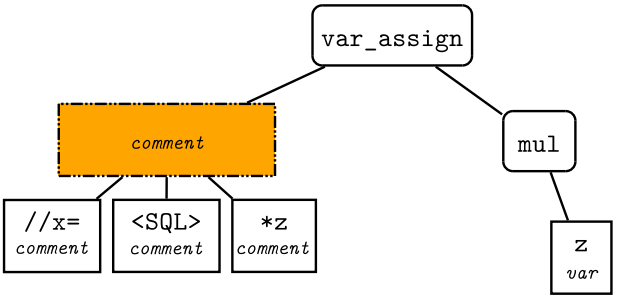
Afterwards, we set `last` to node `//x=`, increase `oldlen` and `newlen`, and retrieve the next elements from both iterators (lines 33–37). The next generated token is `lbox` (i.e. the language box), the next processed node is `=`. The new values of `oldlen` (3) and `newlen` (4) tell us that a node (`=`) was combined into another node, and thus needs to be removed from the parse tree (line 24):



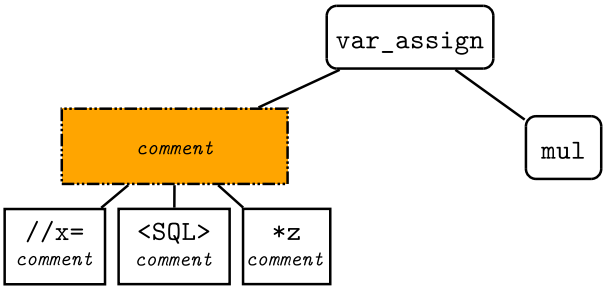
We increase `oldlen` and set the next processed node, the language box, to `old` (lines 25–26). With `oldlen` and `newlen` now both being 4, we yet again end up in the *Overwrite* branch. Since `old` and `new` are the same, updating only changes the node's type. The language box node is then inserted after `last` (i.e. `//x=`), which automatically moves it inside the multinode:



A language box always has a length of 1 so both `oldlen` and `newlen` are increased to 5, which means the *Overwrite* branch is used again. We set `last` to the language box and the next generated token to '`*z`', and the next processed node to `*`. We then overwrite node `*` with '`*z`' and insert it after the language box:



Next we increase `newlen` to 7 and `oldlen` to 6, and set `new` to `None` and `old` to node `z`. We end up in the *Remove* branch of the algorithm and remove node `z`:



Afterwards, both `old` and `new` are `None` and the algorithm terminates.

6.4.2 Merging multiple multinodes

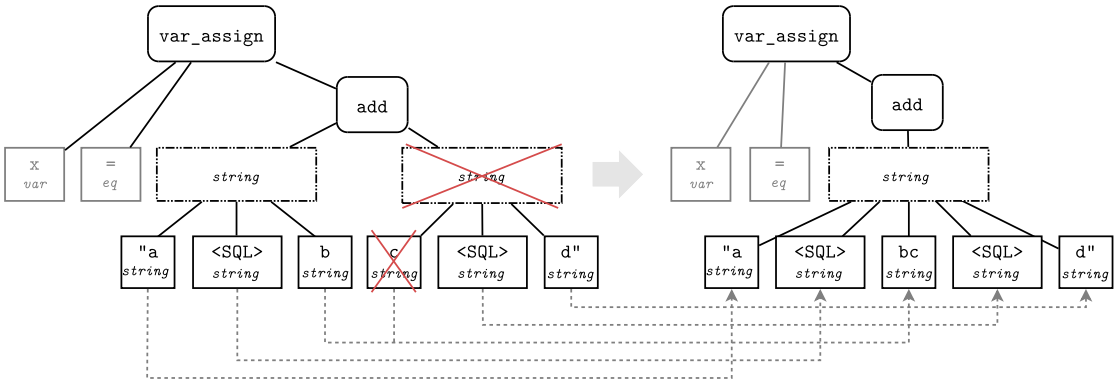
In the following scenario two strings are combined into one. As both strings contain language boxes, and are thus internally represented as multinodes, this example shows the merging of two multinodes. It also highlights how the algorithm avoids destroying and recreating multinodes by reusing existing ones, whenever possible. Let's consider the following program:

```
x = "a  SELECT * FROM table  b" + "c  DELETE FROM table  d"
```

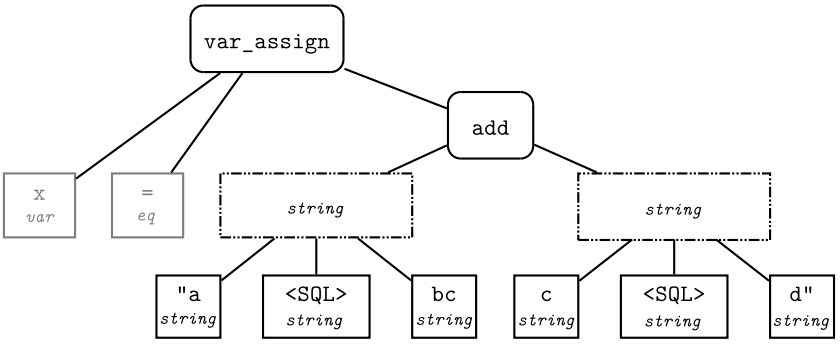
Now we assume the user deletes the two quotes and the plus from the middle, thus joining both strings together. After re-lexing, the lexer generated a single multitoken while the list of processed nodes contains two multinodes:

generated tokens	processed nodes
`(['"a', lbox1, 'bc', lbox2, 'd"'], string)`	`multinode` `multinode`

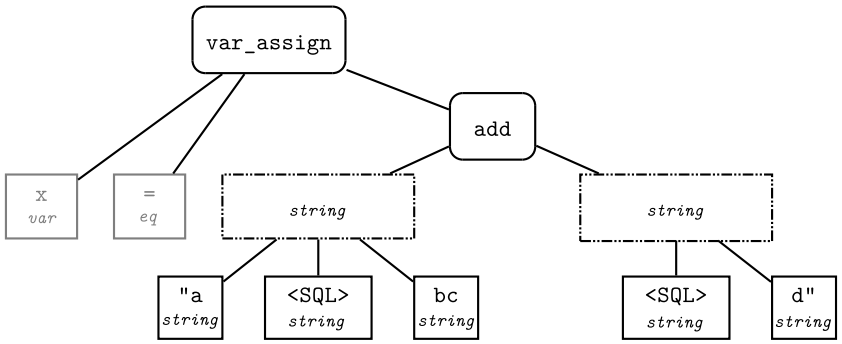
The lexer has merged the two multinodes into one while also combining the two nodes '`b`' and '`c`' into a single token '`bc`'. The following figure shows an overview of how the incremental lexer handles the merging of the two multinodes (nodes untouched by the algorithm are greyed out):



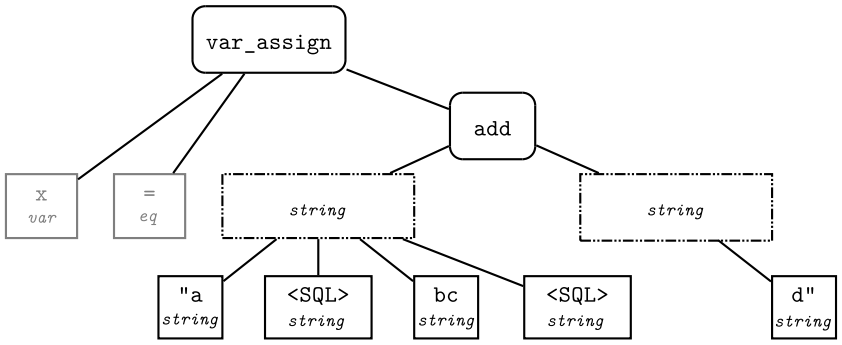
The first two steps of the algorithm use the *Overwrite* branch to overwrite the node `"a` and the first language box (`lbox1`) and move them into a multinode. However, since the first multinode was reused (lines 51–54) and both nodes are already children of it, only their types are updated (line 28). The next processed node is `b` which needs to be overwritten with '`bc`'. Again, since `b` is already inside the target multinode it doesn't need to be moved:



The next generated token is the second language box (`lbox2`) with processed node `c` which is part of the second multinode. The values `oldlen` and `newlen` were increased to 4 and 5, which means we use the *Remove* branch of the algorithm to remove node `c`, which was merged with `b` in the previous step:



Afterwards, `oldlen` is increased to 5 and `old` is set to the second language box, while `new` remains `lbox2`. With `oldlen` and `newlen` being equal, we again use the *Overwrite* branch to update the language box with itself. This time, however, `old`'s parent doesn't match the target `current_mt` (lines 29–30), so the language box is removed from its old parent and moved into the first multinode.



The same applies to the next processed node `d"`, which is removed from its parent and inserted into `current_mt`. Afterwards, since the second multinode is now empty, it is removed from the parse tree (lines 47–48).

6.4.3 Splitting multinodes

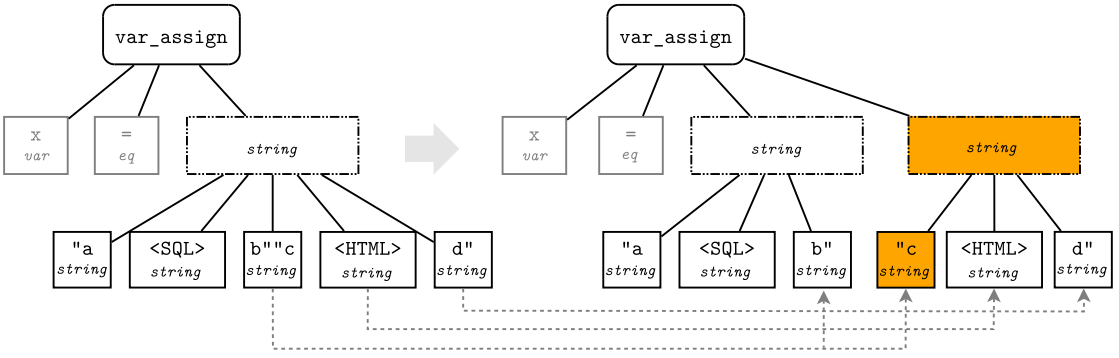
The following example is the reverse of the previous example: a string containing two language boxes is split up causing a multinode to be split into two. This example shows the use of the *Insert* branch, which we have not seen so far. It also shows what happens when two multitokens appear next to each other, which triggers both conditions in line 11 and 14. Let's consider the following program that has a string with two language boxes inside of it:

```
x = "a  SELECT * FROM table  bc  <img src="a.jpg">  d"
```

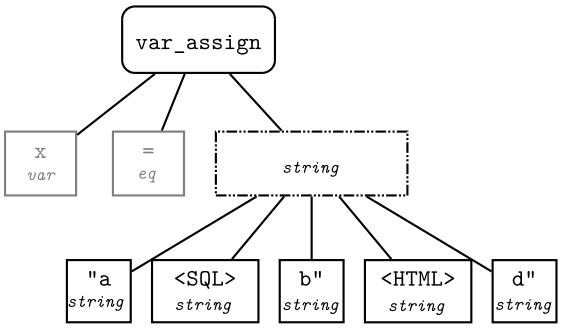
Now the user moves the cursor in-between `bc` and inserts two double-quotes (`""`), thus splitting up the string into two separate strings. After re-lexing, the lexer returns the following generated tokens and processed nodes:

generated tokens	**processed nodes**
`([‘"a’, lbox1, ‘b"’], string)` `([‘"c’, lbox2, ‘d"’], string)`	`multinode`

The lexer has generated two multitokens, both of which contain a language box. This means that the algorithm will have to create an additional multinode for the second string and move some of the nodes from the previous multinode over to it. If the user had only inserted a single quote the algorithm would perform similar to what is described below, except that the nodes after the quote are inserted directly into the parse tree instead of a new multinode. The following figure shows an overview of how the algorithm splits up a multinode into two:

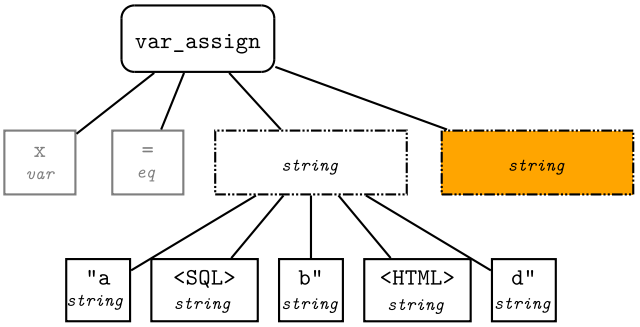


Similar to the previous example, the algorithm begins by reusing the old multinode and overwriting the first three nodes without moving them. While node `"a` and the language box remain the same, node `b""c` is overwritten by ‘`b"`’:

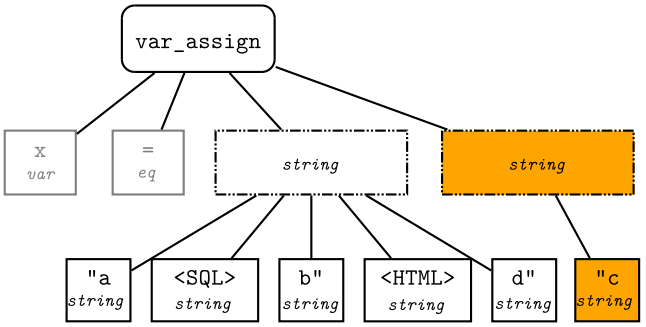


This results in `oldlen=7` and `newlen=5`, with `old` being set to the HTML language box, and `new` being ‘`"c`’. Since ‘`b"`’ was the last child of the multitoken, the multinode is

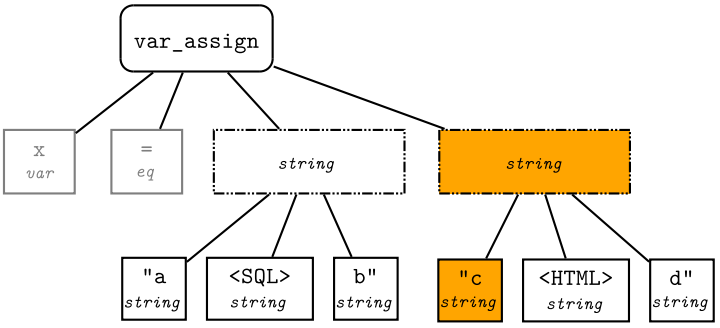
complete, and `current_mt` is set to `None`, while `last` is set to the multinode so that following tokens can be inserted after it (lines 11–13). However, since '`"c`' is also the first child of a multitoken, and `old`'s parent has already been reused, we now create a new multinode and insert it immediately after the old one (line 14–16):



Afterwards, we end up in the *Insert* branch of the algorithm (lines 18–22), in order to insert the new token '`"c`' into the parse tree. However, since the token is the first child of a multitoken, it is inserted into the new multinode instead (lines 39–40):



For the next two generated tokens `lbox2` and '`d"`', we use the overwrite branch again. Although the parent of both remaining processed nodes, `<HTML>` and `d"`, is already a multinode, it doesn't match `current_mt` and so they both have to be moved (lines 29–32):



Chapter 7

Automatic language box detection

This chapter introduces a new algorithm that allows an editor to detect when a user writes code in a language other than the main language. It then automatically creates a language box and moves the written code into it, without the need for any user interaction. The aim is to improve the user's workflow, which otherwise requires them to insert language boxes explicitly and manually. Since many language compositions are ambiguous it is naturally impossible to perfectly predict the user's intentions.[1] Fortunately, it is possible to do a "good enough" job by employing heuristics, which can be improved further by manually tweaking compositions to deal with some language specific edge cases. The chapter is structured as follows: Section 7.1 describes how we can use parsing errors to find automatic language boxes, how we can present multiple solutions to the user, and how automatically inserted language boxes can also automatically be removed again. Section 7.2 describes some of the limitations of automatic language boxes. Section 7.3 describes the implementation of recognisers, which are used during the detection of automatic language boxes. Section 7.4 discusses related work in this area. Appendix F summarises the algorithms shown in this chapter and adds some additional details.

7.1 Using errors to detect language boxes

When reading a composed program it is generally easy for a programmer to tell where a user intended to embed a different language. For an algorithm, however, this is not as simple. Many composition are even so inherently ambiguous that it is, by definition, impossible to determine where one language ends and another begins, for human or machine. Knowing we can't do a perfect job, the challenge is thus to find a 'good enough' heuristic that identifies as many opportunities for the insertion of language boxes as

[1] And sadly the technology to read their minds isn't quite there yet [67].

```
1: def get_customers():
2:     l = SELECT * FROM customers
3:     return [c.name for c in l]
```

(a) Python and SQL

```
1: class C {
2:     public Array powerlist(Array a){
3:         return [x**2 for x in a]
4:     }
5: }
```

(b) Java and Python

Figure 7.1: Two examples showing the attempted composition of languages without the use of a language box. Both compositions result in an error, which can be used to search the surrounding area for potential language box locations. **a)** In this scenario, an SQL language box can be inserted just before the `SELECT` token **b)** Here a Python box can be inserted immediately at the location of the error.

possible. In order to not interrupt the user, this heuristic must be fast. More importantly though, we must minimize the insertion of false positives, which would otherwise force the user to perform unexpected manual work to revert the algorithm's mistakes which is generally considered unacceptable [78, 51, 77].

A good indicator that a user may have wanted to insert a language box is when the current program has a parsing error. Of course, not every error means a language box was meant to be inserted, and not every language box the user wants to insert is guaranteed to produce a parsing error. However, it is a good initial heuristic, since the syntax of two languages is likely to be different, or the two languages share keywords which may clash with each other. This then leads to parsing errors that can be fixed by surrounding the offending code with a language box (see Figure 7.1 for an example).

To decide if a parsing error can be fixed via the insertion of a language box, we need to analyse errors as soon as they appear during the parsing process. This has the advantage that we have additional parsing information available, like the parse stack, to help us decide whether the insertion of a box is a valid and sensible choice. Analysis of the errors is done by the new *automatic language box detector*, which we will call *ALD* from here on. The basic idea is that when an error is encountered, we search for language box candidates whose insertion around the error would render the input successful. The main challenge is therefore in finding suitable candidates.

We start by searching backwards from the error to find locations where a language box would be valid. At each location, and for each language in the composition, we then try to match as much text as possible. If consumed text is valid in one of the languages, a candidate is produced. Each language can produce multiple candidates by consuming more text, even if a candidate for that language has already been produced. Afterwards, each candidate language box is virtually applied to the user's input (replacing the consumed text). If it introduces extra errors or does not fix the initial parsing error, the candidate is discarded. The algorithm for finding candidates is shown in Listing 7.1.

```
1  def find_candidates_D(error: Node):
2    valid_boxes: List[(Language, int)] = []
3    for lang in composition:
4      cut = len(stack)
5      while cut >= 0:
6        if lbox of lang can be shifted at cut:
7          valid_boxes.append((lang, cut))
8        cut -= 1
9
10   candidates: List[(Language, int, Node)] = []
11   for lang, cut in valid_boxes:
12     results = consume_text(stack[cut], lang)
13     for r in results:
14       candidates.append((lang, cut, r))
15
16   f: List[(Language, int, Node)] = []
17   for c in candidates:
18     if confirm_candidate(c) and fixes_error(c, error):
19       f.append(c)
20   return f
```

Listing 7.1: Algorithm for finding language box candidates, for readability divided into several steps. First, for each language l in the composition, we scan the text before the error to find valid locations where a language box of l can be inserted – this step is similar to and was inspired by the finding of isolation nodes, in that it also uses the parse stack to find such locations (lines 2–8). For each of those locations we then try to consume its following text, using l's lexer and parser, and produce a candidate whenever the consumed text is a valid program in l (lines 10–14); `consume_text` stops when we can't consume any more text, because either the remaining text is invalid in l, or there is no more text to consume. Afterwards, we confirm the produced candidates by checking if they fix the previous error and don't introduce any new ones (lines 16–19). Candidates are returned in the order they are found: the closer to the error and shorter they are, the higher they are in the list.

If at the end no candidates are found, the error remains, as normal; if one candidate is found, we automatically insert the appropriate language box; and if multiple candidates are found, we ask the user for help.

If only a single language box candidate is found, the language box can be inserted immediately during the parsing phase. However, we need to make sure that the user can easily undo the insertion of the box, since it may not always be what the user intended to do. In practise it is therefore easier to always insert language boxes after the parsing phase has finished, and re-parse the program with the now inserted box afterwards. This allows us to easily undo automatic insertions by going back one version in the parse tree's history. When the user reverts an automatic language box insertion this way, the editor needs to remember this decision and avoid any further attempts to insert the same language box again, so that the user doesn't have to repeatedly fix the algorithm's mistake.

7.1.1 Finding candidates

As an example, let's consider a composition of Java and Python, where Python functions are allowed wherever Java functions are valid. For this we embed into Java a subset of the Python grammar, accepting only Python functions, which we call `pyfuncdef`[2]. We now assume the user wrote the following program:

```
1   class Example {
2       def x():
3   }
```

This program currently has a parsing error at '`:`'[3]. We now try to find a valid location on the parse stack, where a `pyfuncdef` language box can be inserted. Such a location is just before '`def`'. In the next step we try to find candidates by consuming text, starting at '`def`'. The language `pyfuncdef` can consume everything up to, but excluding '`}`', which causes a parse error and thus cannot be consumed. The only consumed text up to that point is '`def x():`', which is not a valid program in `pyfuncdef`, and so no candidate is produced as a result.

Assume now that the user inserts '`pass`' into the above program, resulting in:

```
1   class Example {
2       def x():
3           pass
4   }
```

While the error remains at '`:`', more text can be consumed this time. Again, we use the language `pyfuncdef` to consume text starting from '`def`'. Upon consuming '`pass`', a valid Python function has been found and a candidate is created. We continue consuming text, to find further candidates. However, as before, we stop upon reaching '`}`' which is not valid in `pyfuncdef`. Having found a candidate, we now need to check if it fixes the error and doesn't introduce any other errors immediately following the box, when it's applied to the users input. We replace the consumed text with a language box of `pyfuncdef` and attempt a parse in the outer language, Java, starting at the location of the language box. As soon as we successfully parsed '`}`' we can stop, confirming that the insertion of the candidate doesn't introduce any new errors (Section 7.1.2 discusses why it is sufficient to stop parsing after '`}`' and why we don't have to parse the entire program to determine if the candidate is valid). Finally, we check if the original error has been eliminated, which is the case here, as the error node '`:`' was included in the candidate language box. The candidate is thus valid and can be automatically inserted into the program.

[2]This grammar can be constructed from the Python grammar (shown in Appendix G) by changing the start rule to `funcdef`.

[3]The parser expected an opening bracket here, as Java parses '`def x()`' as a function definition '`x`' of the type '`def`' which needs to be followed by opening and closing brackets.

```
1: class X {
2:     public void x(){
3:         int x = [1,2,3] *;
4:     }
5: }
```

```
1: class X {
2:     public void x(){
3:         int x = [1,2,3] *;
4:     }
5: }
```

Figure 7.2: An example showing why parsing further than necessary to confirm a language box candidate, can lead to the candidate being falsely rejected. The example shows a composition of Java and Python, that allows Python expressions wherever Java expressions are valid. The user inserted a Python list into the program (left), which produced a language box candidate (right). To validate the candidate we insert it into the program and parse it. However, after parsing '*', an error occurs which would invalidate the constraint that an automatic language box must not introduce new errors, and thus reject the candidate, despite this being an otherwise valid insertion.

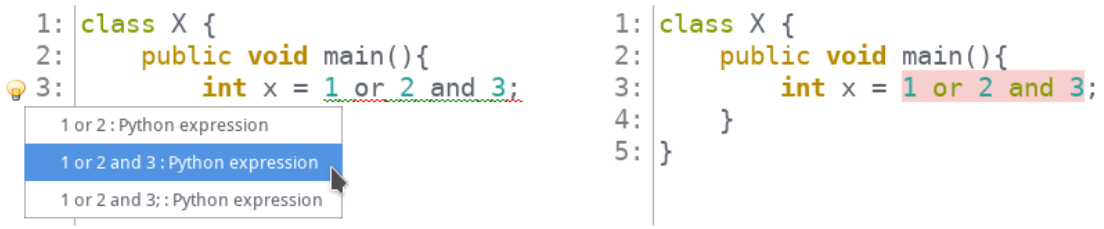
7.1.2 Confirming candidates

In order to confirm if a candidate is valid, we need to check that it does not introduce new errors when it is inserted into the parse tree, and also that it fixes the initial parsing error. In order to determine if a candidate doesn't introduce new errors, we can simply check if the first non-whitespace token that follows it can be shifted. The reasons for this are twofold. Firstly, we don't want to parse too far past the language box for performance reasons, and because this could lead to candidates being falsely rejected (see Figure 7.2 for an example). Secondly, whitespace tokens, which most programming languages use, have the property, that they can almost always be shifted after another terminal symbol. This means that a whitespace token following a language box candidate would always confirm the candidate, even if the insertion of the box would cause an error immediately after that whitespace. This can sometimes lead to invalid candidates being accepted (see Figure 7.3 for an example).

7.1.3 Handling multiple solutions per language

While many compositions only have one valid outcome, sometimes there are multiple options for the insertion of a language box. This can be either that there are multiple languages that can be inserted, or that there are multiple variants of consumed text for a single language. An example for the latter is shown in the following composition of PHP and Python, which allows any Python box wherever a PHP expression is valid.

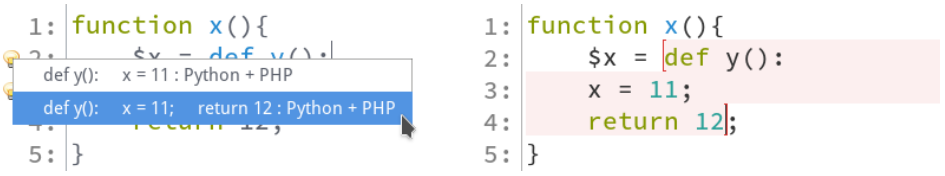
```
1: function x(){
2:     $x = def y():
3:     x = 11;
4:     return 12;
5: }
```



(a) Using first token **(b)** Using first non-whitespace token

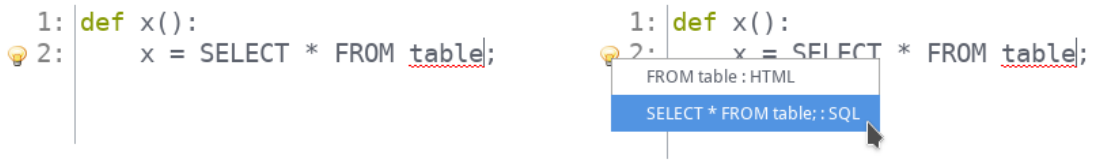
Figure 7.3: An example showing how using only non-whitespace tokens to confirm language box candidates gives better results. The example uses a composition of Java and Python, which allows Python expressions in place of Java expressions. The user just pasted the Python code '`1 or 2 and 3`' into the assignment, in between '`=`' and '`;`'. **a)** If we use any first token to confirm candidates, we get three results. The first option, '`1 or 2`', is a valid candidate, because the following token is whitespace, which can be shifted, confirming the candidate; however the following '`and`' then causes an error. The same is true for the third option, '`1 or 2 and 3;`', which is followed by a newline, which confirms this candidate, even though the '`}`' then errors because the semicolon of the assignment was included in the Python language box. **b)** When using the first non-whitespace token for candidate confirmation, we reduce the three options to one, which can then be automatically inserted.

In the example there are two options to fix the error at '`y`' in line 2, via the insertion of a Python box. We can include everything from '`def`' up to, and including, '`return 12`', or we can stop after '`x = 11`'. Since both options are valid and fix the error, we can't automatically pick one. Instead, we display an indicator that there are multiple solutions available (in *Eco* this is done via a light bulb icon next to the line numbers). From there the user can choose one of the solutions from a drop-down list. Clicking on one of the options then automatically inserts a language box and replaces the relevant code:



7.1.4 Limiting automatic insertions

Some grammars can be very unrestrictive, allowing almost any text to be valid. In HTML, for example, any combination of characters that does not contain '`<`' or '`>`' is a valid program. Unfortunately, this means that in a composition with HTML most errors can be solved by simply wrapping them with a HTML language box, often resulting in something the user didn't intend. For instance, the following example shows a program written in a composition of Python that allows SQL and HTML wherever Python expressions are valid:



We can see that the error at '`table`' can be solved by either wrapping '`SELECT * FROM table`' into a SQL language box, or by just wrapping '`FROM table`' into a HTML language box. While a human can easily guess that the user meant to insert a SQL box here, this scenario is impossible for the algorithm to detect. To remedy this, the creator of the language composition can add hints that limit the insertion of some language boxes in such cases.

There are two ways in which a language author can add limitations to a composition: they can define the symbols that an automatically inserted language box must start with; or they can exclude specific symbols that must not appear at the beginning of a language box. *Eco*'s language composition interface thus provides two functions `set_auto_include(lang, tokens)` and `set_auto_exclude(lang, tokens)`. Both take as input the sublanguage which we want to limit and a list of tokens that are to be allowed or disallowed at the beginning of the language box. We can then solve the above problem, by either limiting automatic HTML language boxes to HTML tags, e.g. '`<img`', using `set_auto_include("HTML", "tag_open")`. Or we can disallow HTML language boxes to start with normal text, using `set_auto_exclude("HTML", "text")`.

Of course, another valid solution is to simply define a more fine-grained composition. For example, instead of creating a Python/HTML composition that allows all HTML to be valid wherever Python expressions are valid, the composition could be restricted to only allow HTML-tags (e.g. '`<img`', '`<html`', etc) by embedding only a subset of the HTML grammar. Although, this may not always be possible, for example if the grammar doesn't separate rules on such a fine grained level.

7.1.5 Automatically removing boxes

Despite the solutions described in 7.1.4 it is impossible to always predict where language boxes should be inserted and there will be some cases where automatically inserted language boxes do not match the user's intentions. While the user can always undo a wrong insertion by pressing Ctrl+Z, we do not want to burden them with the task of repeatedly cleaning up the algorithm's mistakes. The following composition of Python and SQL, though admittedly a rather unlikely scenario, exemplifies the problem:

```
1   def x():
2     SELECT = 1
3     FROM = 2
4     table = 3
5     x = SELECT * FROM table
```

After typing in this program, an SQL language box will have been automatically inserted, replacing the text '`SELECT * FROM table`'. From a syntax perspective, this was a valid decision. However, we can see from the variable declarations that the user's intention was to multiply the three poorly named variables and simply forgot a '`*`' between '`FROM`' and '`table`'. Instead of requiring the user to manually undo the insertion, *Eco* can automatically remove inserted language boxes again once more information has become available. In this example, once the user inserts the missing '`*`' between '`FROM`' and '`table`', the SQL language box becomes invalid. This gives us a good clue that the language box was inserted accidentally and needs to be removed; with the restriction, that its entire content must be valid in the outer language. However, sometimes an automatically inserted language box can become valid in both the inner as well as the outer language, after the user made additional changes. In those cases *Eco* prioritises the outer language and removes the box, but only if the boxes contents *and* its surrounding context can be parsed in the outer language. The following constraints summarise the above (an example using these constraints is given in Figure 7.4):

1. Only automatically inserted boxes can be automatically removed again.
2. If the language box content is invalid, it will be removed only if its content can be parsed in the outer language.
3. If the language box's content is valid in both the inner and outer language, it will be removed only if its removal doesn't introduce new errors (i.e. if its contents *and* the first non-whitespace token following it can be parsed).

Using these constraints for the example from before, the inserted box would be removed as soon as the user inserts a '`*`' between '`FROM`' and '`table`'; however, it stays if more SQL code is added even it that makes the SQL code temporarily invalid. Note that language boxes that were inserted by the user, via choosing one of the suggested candidates, count as manual insertions and won't be automatically removed, even if they become invalid.

Figure 7.4: An example showing the constraints for automatic language box removal in practice. The example uses a composition of PHP and Python where Python expressions are valid at PHP's top-level. We can see that as the user types, language boxes are getting automatically inserted and removed depending on their content. At the beginning the user types '`f`', which is not valid in PHP and thus gets replaced by a Python language box, making the program valid again. The box stays until the user inserts '`(`' which makes the Python language box invalid, and since all of its content can be parsed in the outer language, the box is removed. As soon as the user inserts the closing bracket '`)`', a Python box is inserted again to fix the PHP parsing error caused by a missing semicolon. When the user eventually inserts the missing semicolon the contents of the language box are valid in both PHP and Python. However, since the box can be removed without introducing an error, we prioritise the outer language, and remove it again.

7.2 Limitations

This section highlights the limitations of the heuristic used, by giving some examples of ambiguities that cannot be resolved with automatic language boxes, resulting in unwanted language boxes or no language boxes being inserted at all.

7.2.1 No detection

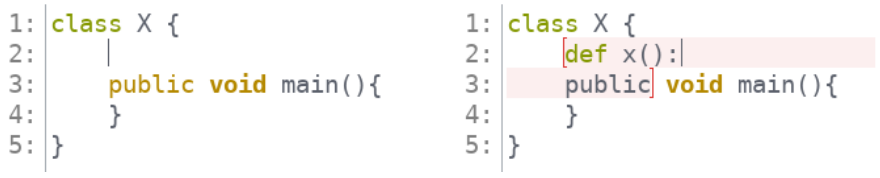
One example, where error based detection of automatic language boxes doesn't work, is when the outer language can match everything in the inner language. For example, HTML can match arbitrary text between tags. This makes compositions, where other languages are embedded into HTML, impossible to detect automatically. The following code example shows this:

```
1 <html>
2 import sqlite
3 sqlite.connect("test.db")
4 </html>
```

Since HTML allows any text in between its tags, the insertion does not cause an error, and thus the *ALD* is never called to detect automatic language boxes. However, even to a human it is unclear if the user meant to insert a Python box, or simply wanted to print out the code in HTML. A non-error based heuristic for finding language box candidates could show the user that a language box would be valid here, by constantly trying to match language boxes at each token location during parsing. However, such a solution would be much slower than the error based approach and would produce a large amount of candidates at various locations that they user may not care about.

7.2.2 Wrong detection

Another problem is when the insertion of an automatic language box extends into parts of the outer language, even if those parts weren't meant to be included in the box. For example, this can occur when composing Java and Python, allowing Java functions to be replaced with Python functions, as shown below:



The user attempted to insert a Python function above the Java function. However, as the user was typing, a box was automatically inserted which extended into the Java function, and used the keyword '`public`' as the body of the Python function. This is valid, since the keyword is optional in Java, so the insertion of that box does not create any errors, even though it is clearly not what the user intended.

This problem can be mitigated using additional information stored in the parse trees. Since we have access to the entire history of the program, we know the order in which tokens were inserted. We can use this to add a restriction to language boxes which disallows the inclusion of tokens which are older than the first token in the language box. In this example, the '`public`' token was entered before the user started typing '`def x():`'. Adding this restriction would thus avoid the above problem and only insert a language box when the user adds a body to the Python method definition. The downside of this solution is that code intended for a language box cannot be typed out-of-order, which some users may want to do. We thus can't add this limitation to the heuristic by default, and instead need to make it optional, so that it is only used for grammars, where this is a problem (e.g. in *Eco* this can be done by setting the flag `auto_limit_new` of a composition to `true`).

7.2.3 Grammar limitations

Sometimes the structure of a grammar can be responsible for the failure to generate automatic language boxes. For example, let's consider a composition of PHP and Python, that allows Python expressions wherever PHP expressions are valid. The user then edits the composition as follows:

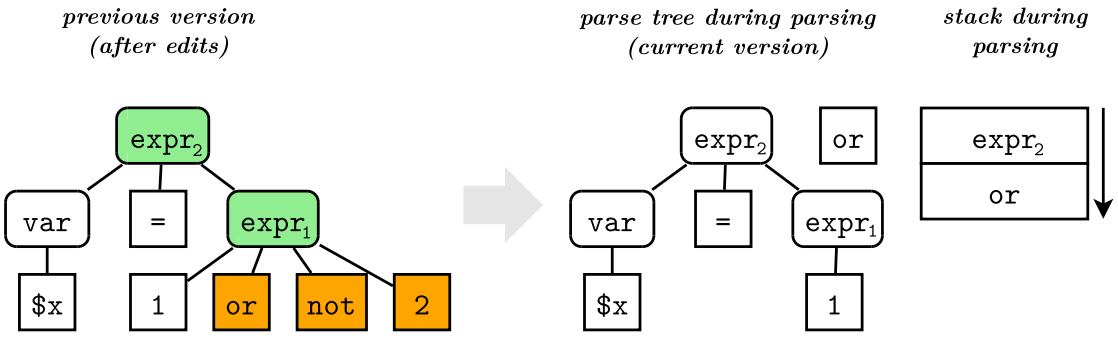


Figure 7.5: An example showing how sometimes the grammar of a language can keep the automatic language box detection from finding appropriate candidates. Shown on the left is the elided parse tree after the user edited the PHP expression '`$x = 1;`' by inserting the Python code '`or not 2`'. However, we can see on the right, that by the time the error occurs, PHP has already reduced '`$x = 1`' to $\texttt{expr}_2$, and shifted `or` onto the stack. The only valid locations for a language box we can thus find are before $\texttt{expr}_2$ and '`or`'. In particular, we cannot find the location before '`1`' which would allow us to wrap the entire expression into a box.

The user might reasonably expect to have a second option here, which would allow the content '`1 or not 2`' to be wrapped into a Python language box. The algorithm, however, couldn't detect this and instead only found a single candidate, '`not 2`', which was automatically inserted. The problem lies within the PHP grammar, which also has a keyword '`or`', and how the location for language box candidates are calculated. As described earlier in this chapter, we use a technique similar to error recovery, that rewinds the parse stack to find a position on the stack, where the language can be inserted. However, the way the PHP grammar is structured and thus parses the above input, makes it impossible to find a location on the stack that allows '`1 or not 2`' to be wrapped into a language box, as Figure 7.5 shows.

7.3 Recognisers

This section describes in more detail how language box candidates are constructed and how we can improve performance by not using a full incremental parser to find language box candidates, but instead using a simple batch recogniser. A recogniser is a parser that doesn't execute any actions or generates a parse tree, but rather only validates the input. This section also shows how a recogniser can be used to decide whether or not a language box can be removed again.

Section 7.1 described, how in order to find language box candidates, we try to consume as much text surrounding the error node as possible, generating a candidate every time the surrounding text is valid in the language of the box. Since we are only interested in the text that can be parsed and don't need to generate a parse tree from it, a simple batch

```
1  def consume_textD(node: Node, lang: Language) -> List[Node]:
2    lexer = lexer for lang starting at node
3    parser = parser for lang
4    results: List[Node] = []
5    token = lexer.next_token()
6    while token is not None:
7      if parser.parse(token):
8        if parser can accept current input:
9          results.append(lexer.last_node)
10     else: # parse error
11       break
12     token = lexer.next_token()
13   return results
```

Listing 7.2: An algorithm for consuming text and returning valid substrings for the creation of language candidates. We first create a lexer that produces tokens in language `lang`, starting at `node` (line 2). We also create a parser for `lang` (line 3) which is used to parse input (line 7) and test if the input parsed so far can be accepted (line 8); the latter can be achieved by simply pretending that the next token is the end-of-file token, and test if the parser reaches an accept state. We then consume as much text as we can, producing tokens and parsing them (line 6–12), until no more text can be consumed, either due to an error (line 10), or because we've reached the end of the input. Each time a token was successfully parsed, we test if the input parsed so far is valid (line 8), and if so, store the last node from the original parse tree that the lexer processed (line 9). This node is later used to derive the substring from which the language box is created.

recogniser is a good alternative to a full parser, as it improves performance and reduces memory usage. An important restriction is that the recogniser must lex and parse input from the parse tree without actually altering the parse tree or any nodes within it. A recogniser in the *ALD* takes as input a node from which it starts parsing, and returns all valid substrings from that input, until either a parsing error occurs, or there is no more input left to parse. Substrings are returned in form of their end node in the original parse tree. The substring can then be derived by reading all tokens between the start and end node. The algorithm is shown in Listing 7.2.

7.3.1 Custom recogniser for Python

In order to produce language box candidates for whitespace-sensitive languages like Python, we need to be able to create indentation tokens during the consumption of input. Section 4.2 described how support for whitespace-sensitive languages was added by allowing an additional phase between lexing and parsing that inserts indentation tokens into the parse tree. This method can't be used here, since we do not know yet where the input ends. Fortunately, since the recogniser is not incremental (and doesn't need to be), this is not necessary and indentation tokens can be generated on the fly. To do this, we implement a separate recogniser for Python which uses a wrapper around the lexer that inspects the tokens that the lexer produces and, when appropriate, returns indentation

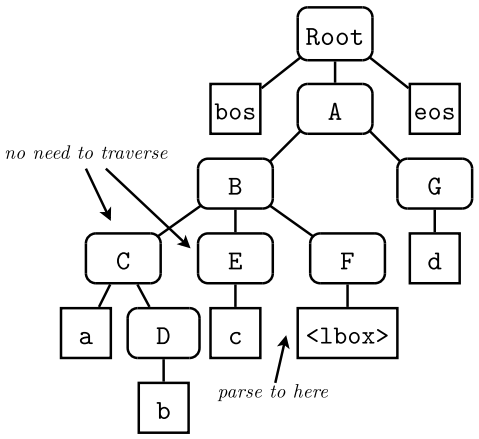
```
1  def next_token_D(todo: List[Token], indent: List[int]) -> Token:
2    if todo:
3      return todo.pop(0) # return first element
4    token = lex.next_token()
5    if token is newline:
6      prevl = previous line
7      currl = current line
8      if prevl is not empty:
9        todo.append(NEWLINE)
10     if currl is not empty:
11       if prevl.ws < currl.ws:
12         indent.append(currl.ws)
13         todo.append(INDENT)
14       elif prevl.ws > currl.ws:
15         ws = indent.pop()
16         while ws > currl.ws:
17           todo.append(DEDENT)
18           ws = indent.pop()
19         if ws != currl.ws:
20           todo.append(UNBALANCED)
21   todo.append(token)
22   return todo.pop(0)
```

Listing 7.3: To support language box detection for Python, we create a method that wraps around the lexer and produces indentation tokens as needed by keeping track of the indentation levels. Producing and returning indentation tokens delays the parsing of tokens which have already been lexed. In order to not forget to parse those tokens, we use a to-do list (line 3). We always return the first item of that list. If it is empty, we continue lexing from the input (line 4). The remaining code is similar to the batch indentation approach described in Section 4.2.1.

tokens instead. While the Python recogniser inherits most of the behaviour of the default recogniser, it also needs to keep track of the indentation levels. Listing 7.3 shows how the recogniser wraps around the lexer and produces indentation tokens whenever they are needed.

7.3.2 Incremental Recogniser for auto removal

Section 7.1.5 described how automatically inserted language boxes can automatically be removed again, if they become invalid. The condition for this is that the box's content is valid in the outer language. In essence, we need to temporarily remove the language box, paste its content where the box was before, and then check if the program can still be parsed; of course without actually altering the parse tree. A recogniser is thus again a good choice. In order to test if the contents of the box are valid in the outer language, we first need to initialise the recogniser to the state just before the language box would be parsed. We can do this incrementally, by only traversing subtrees that are direct ancestors of the language box. All other subtrees can be skipped (i.e. incrementally shifted) similar to the default incremental parser.




```
1  def preparse_D(bos: Node, lbox: Node, parser):
2    # "mark" ancestors of lbox
3    path_to_lbox = set()
4    parent = lbox.parent
5    while parent is not Root:
6      path_to_lbox.add(parent)
7      parent = parent.parent
8
9    # initialise parser
10   node = bos
11   while node is not lbox:
12     if node not in path_to_lbox:
13       parser.parse(node)
14     else: # traverse node
15       if node.children:
16         node = node.children[0]
17       else:
18         node = node.right_neighbour()
```

Figure 7.6: A parse tree with a language box that we want to automatically remove (left) and the preparse-algorithm (right). In order to test if the box's content is valid in the outer language, we initialise a recogniser to the state just before we would parse the language box, using the recogniser's `preparse` method. This is done, by 'marking' the ancestors of the language box, as if the language box contained changes (lines 3–7), and then incrementally parse those subtrees up to the box (lines 10–18).

This process is implemented via an additional method `preparse`, which incrementally initialises a recogniser's parsing state to the state just before a given node. Figure 7.6 shows an example of a parse tree, where we want to test if a language box can be removed, and the algorithm to initialise the recogniser. After the recogniser has been initialised, we simply use `consume_text` to parse the content of the language box. If the entirety of the language box can be successfully parsed in the outer language, the box can be removed. Of course, depending on the parsing status of the box, we may also need to parse at least one non-whitespace token following the former language box.

7.4 Related work

Scannerless parsing [86, 79] is well suited for language composition since it can parse any context-free grammars which are closed under composition. A notable example of this is Spoofax [41], a language workbench for extending and composing context-free grammars. Although scannerless parsing can parse ambiguous programs by creating a parse forest, it is still necessary to reduce the parse forest into a single parse tree in order to compile or interpret it. A common way to reduce the parse forest is to remove all parse trees that are invalid when considering additional information about the input, such as types [84]. Using type information can avoid the need for separators to disambiguate languages in many cases, however still requires a set of disambiguation rules, such as

preferring identifiers in the outer language (i.e. the meta language) over those in the inner language (i.e. the object language), or choosing the shortest path if multiple valid options are available. Unfortunately, not all ambiguities can be solved this way, making the use of separators still a necessity.

Choosing separators is not trivial, since they can still introduce ambiguities if the same separator symbol is used for different nonterminals of the embedded language. This requires *explicit disambiguation*, i.e. using a different separator for each embedding with a different nonterminal symbol, e.g. `cls{...}cls` for embedding classes and `func{...}func` for functions [8]. Since this can quickly cause "syntactic clutter" [13, p. 4], an alternative solution is to use type information to automatically disambiguate embeddings, thus reducing the amount of unique separators that would otherwise be required [91, 13]. Unfortunately, since these approaches are dependent on types, they do not work for dynamically typed languages. Furthermore, heuristics such as picking the shortest path on multiple valid options can hide options the user cares about (as shown in Section 7.1.3).

Despite these improvements, Spoofax's grammar definition SDF, which allows arbitrary CFGs, can give no guarantees that its grammars are unambiguous, even more so when they are composed together, since composing two unambiguous grammars can lead to an ambiguous one. Spoofax thus uses reject grammars, which solve this problem, but make such grammars context-sensitive [80].

Another notable example for language composition is Copper [82] which implements a context-aware scanner to solve ambiguities in language compositions. The basic idea is that the parser can tell the scanner which tokens it can parse next and the scanner can only return results from that list. This means that if two composed languages have similar tokens (e.g. keywords, identifiers), the lexer solves ambiguities by returning the token for the language that it is currently parsing (i.e. that is currently in context). This allows Copper to compose languages by extending the host language's grammar rules with references to rules in the embedded language. However, at the point where another language can be embedded, i.e. where the two languages meet, a token may be valid in both the host as well as the embedded language. Copper solves this problem via `dominates` clauses which are defined within the lexing rules. For example, if an identifier in the host language clashes with a keyword in the embedded language, then we can define a clause that says that the keyword dominates the identifier and needs to be prioritised. Unfortunately, this has the downside that those tokens cannot be used in the host language at that point, restricting the host language's expressiveness. This also means that each composition needs to determine all of those cases and modify grammar and lexer in ways that solve these ambiguities, making it impossible to create a one-size-fits-all solution for arbitrary compositions.

Chapter 8

Conclusion

8.1 Summary

In this thesis I first explained and corrected Wagner's incremental parsing algorithms, and then extended them with the concept of incremental ASTs and support for whitespace-sensitive languages such as Python. Using the revised algorithms as a basis for language composition editing, I introduced language boxes which allow the fine grained composition of different languages, solved the problem of language boxes within tokens such as comments and strings, and added functionality to automatically detect and insert language boxes wherever possible and feasible. All techniques presented in this thesis were implemented in a reference prototype editor *Eco*. A more detailed summary is as follows.

Chapter 2 gives a background into incremental lexing and parsing, by introducing and expanding the techniques developed by Tim Wagner. During the development of *Eco* incremental parsing techniques have not only been shown to be useful for language composition, but have enabled the quick implementation of IDE features (e.g. syntax-highlighting) that would have been substantially more work using traditional parsing techniques. Unfortunately, Wagner's description of managing the history of parse trees left out many details for the reader to fill in so Chapter 2 also showed my own algorithms to store and recover parse trees which were based loosely on Wagner's ideas.

In Chapter 3 I summarised and corrected Wagner's error recovery algorithms. These techniques allow the incremental parser to integrate user changes into the parse tree that would otherwise be hidden by errors. This results in more complete ASTs at earlier stages of the program, and additionally produces useful error messages based on the actual edits the user made. However, while a history-based error recovery approach works well in most

cases, for some the produced error messages are less helpful and I therefore recommend highlighting the actual parse error alongside the erroneous user edits.

I then extended Wagner's incremental parser by adding support for incrementally generated abstract syntax trees as well as incremental handling of indentation based languages such as Python (Chapter 4). Incremental abstract syntax trees fit quite naturally within the incremental parsing techniques and only minor edits were necessary to extend the incremental parser. Incremental ASTs not only allowed more accurate syntax-highlighting that isn't plagued by some of the issues that regular expression based approaches seem to have; it also gives us a basis on which we can run semantic analyses of the program as it is being typed without having to revert to subprocesses which only run occasionally and are thus often out-of-date by the time they are finished.

I then showed how an editor can be extended with language boxes using the editor *Eco* which was developed alongside this thesis (Chapter 5). I showed how language boxes simplify the composition of languages in a manner unimaginable when using existing approaches which often have to revert to unsightly separators between languages and require careful construction of the composed grammars. I also briefly showed how language boxes can be easily extended to be used for non-textual languages, an addition that opens the door for more research and many creative compositions. *Eco*'s use within two case studies shows that it is mature enough to be used in practice, despite it only being a prototype with relatively little development time compared to other IDEs. The PyHyp case study also showed how language boxes can solve the code migration problem and one can imagine this scaling to larger projects as well.

While introducing language boxes into an editor is straight-forward, it has one challenging knock-on effect. Since language boxes are allowed anywhere within a program, they can be commented out or even inserted into strings, something that traditional lexers are not expected to handle. I therefore presented a solution for such scenarios by introducing multinodes (Chapter 6), essentially a special token that can contain multiple other nodes, while presenting itself to the parser as a normal token. This allows the editor to embed language boxes inside of strings or comments without the need to flatten them and thus irrecoverably destroying their internal structure.

Language boxes need to be inserted manually and explicitly, which sometimes can become a tedious, and in some obvious cases unnecessary, task. I thus extended the editor with automatic language boxes (Chapter 7). This approach uses error information to detect when a user attempted to insert a language box and then finds and inserts the correct box for them automatically. In the same vein automatically inserted language boxes are also removed again if their content has changed in such a way that makes it clear that it is

meant for the outer language. Implementing those techniques into *Eco* have shown them to work well in many cases making the use of language boxes more intuitive and natural.

8.2 Future Work

While this thesis showed how to develop a basic incremental parser-based text editor from the ground up, *Eco* is still far from being a full IDE. One of *Eco*'s main issues is performance. While the editing of programs is fast, thanks to the incremental parser, compiling and loading cached grammars is slower than it needs to be. *Eco* was planned and developed as a prototype to create and test the techniques described in this thesis. Thus, performance took a back seat to ease of implementation and debugging. For these reasons, *Eco* was developed in Python making heavy use of object-orientation; switching to low-level data structures could thus greatly improve speed and memory usage. I thus believe that an important and useful future work area would be a rewrite of *Eco* in a compiled language.

A further area of research is semantic analysis. Chapter 5 briefly mentioned the use of name binding to highlight unused variables and to implement a basic form of code completion. Currently, this requires the analysis of the entire AST, which is too time consuming to run each time the AST changes (which is after every keypress) and is thus currently turned off by default. A common solution in most IDEs, which have the same problem, is to run name binding in a separate process in the background. However, I believe that name binding can be made incremental using the already incremental AST. This would allow us to update name binding information instantly as the user types the program. It would also improve code completion, which in many IDEs has a noticeable delay. As a foundation for incremental name binding we could use the work of Néron et al. [61] who create scope graphs from ASTs to connect references to declarations. Since each node in the graph relates back to a node in the AST, I believe it to be possible to generate incremental scope graphs by updating nodes in the scope graph whenever their relating AST node changes.

In Chapter 5 I briefly showed the use of non-textual languages in *Eco*, albeit the usefulness of the given examples is somewhat limited. I believe, however, that this area opens up plenty of research opportunities to see how different and more interactive non-textual languages can be integrated into an editor. For example, one possible integration would be tables or databases, which could be manipulated and referenced from within the editor. Another is the embedding of graphs into the code that implements them.

Currently, *Eco* only uses Wagner's non-repairing history-based approach for error recovery shown in Chapter 3. This solution works well for smaller edits where it can precisely isolate and show the source of such errors. However, due to the lack of a history, this doesn't work on erroneous files that are directly imported or pasted into *Eco*. Such scenarios still require a traditional repairing error recovery approach. Additionally, history-based error recovery can sometimes be misleading if the change that caused the error was valid and the error happens later on. These issues, paired with the difficulty of implementing the history-based error recovery algorithms (see Chapter 3), make it hard to justify implementing this approach over traditional error recovery solutions. A further area of research would thus be investigating if traditional error recovery solutions can be made incremental and embedded into an incremental parser. A possible candidate for this is *MF* [20], a repairing error recovery algorithm based on the Fischer [28] family of error recovery algorithms, which can find the complete set of minimum cost repairs in an acceptable time and reduces the cascading error problem.

Habits are hard to break and syntax-directed editors have shown that convincing programmers to switch to a different tool than what they've used for years, can be difficult. For most programming languages, which are largely text-based, this is not a huge problem, as they are compatible with any text editor. Unfortunately for *Eco*, its support for language composition requires programs to be represented as parse trees internally as well as when they are saved to disk, as language boxes are difficult to represent textually. This means that composed programs written in *Eco* can currently only be read and edited in *Eco*. One way this problem could be solved is to allow other text editors to plug into *Eco*: while the text editor takes care of the rendering, it communicates with an *Eco* background process, forwarding any user input and letting *Eco* handle everything related to parsing.

Another significant problem is version control. Since *Eco* programs are stored as parse trees, such files are incompatible with version control systems such as Git or Mercurial. While a temporary solution is the exporting of those files into plain text, this is only useful to see, at a quick glance, what was changed, but doesn't display language boxes, and more importantly can't be used for patching or version control since language box information is lost during the export. However, this problem can be solved by using diffing algorithms for trees, such as GumTree [25]. Since language boxes behave like all other nodes with children, these algorithm can be used for *Eco* files without modification.

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, Principles, Techniques. *Addison Wesley*, 7(8):9, 1986.

[2] Luís Eduardo de Souza Amorim, Michael J Steindorfer, Sebastian Erdweg, and Eelco Visser. Declarative Specification of Indentation Rules: A Tooling Perspective on Parsing and Pretty-Printing Layout-Sensitive Languages. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, pages 3–15. ACM, 2018.

[3] Tom Anderson, Jim Eve, and James J. Horning. Efficient LR(1) parsers. *Acta Informatica*, 2(1):12–39, 1973.

[4] Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):547–576, 1986.

[5] Robert A Ballance, Susan L Graham, and Michael L Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):95–127, 1992.

[6] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Fine-grained Language Composition: A Case Study. In *ECOOP*. Dagstuhl LIPIcs, July 2016.

[7] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Unipycation: A Case Study in Cross-language Tracing. In *VMIL*, pages 31–40, Oct 2013.

[8] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203)*, pages 143–153. IEEE, 1998.

[9] Jean-Philippe Bernardy. Lazy functional incremental parsing. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 49–60. ACM, 2009.

[10] Patrick Borras, Dominique Clément, Th Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. *Centaur: the system*, volume 13. ACM, 1988.

[11] Marat Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Master's thesis, University of California, Berkeley, Jun 2001.

[12] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52 – 70, 2008.

[13] Martin Bravenboer, Rob Vermaas, Jurgen Vinju, and Eelco Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *International Conference on Generative Programming and Component Engineering*, pages 157–172. Springer, 2005.

[14] Max Brunsfeld. Treesitter. `http://tree-sitter.github.io/tree-sitter/`, 2018. Accessed: 2018-01-04.

[15] David G. Cantor. On the Ambiguity Problem of Backus Systems. *J. ACM*, 9(4):477–479, Oct 1962.

[16] Thomas E. Cheatham. Motivation for Extensible Languages. *ACM SIGPLAN*, 4(8):45–49, Aug 1969.

[17] Rafael Corchuelo, José Antonio Pérez, Antonio Ruiz-Cortés, and Miguel Toro. Repairing syntax errors in LR parsers. 24:698–710, November 2002.

[18] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190 – 210, 2006.

[19] Franklin Lewis DeRemer. *Practical translators for LR (k) languages.* PhD thesis, Massachusetts Institute of Technology, 1969.

[20] Lukas Diekmann and Laurence Tratt. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. arXiv 1804.07133, January 2019.

[21] Veronique Donzeau-Gouge, Gerard Huet, Bernard Lang, and Gilles Kahn. *Programming environments based on structured editors: The MENTOR experience.* PhD thesis, Inria, 1980.

[22] Patrick Dubroy and Alessandro Warth. Incremental Packrat Parsing. In *SLE*, pages 14–25. ACM, 2017.

[23] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[24] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In *International Conference on Software Language Engineering*, pages 244–263. Springer, 2012.

[25] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[26] Manuel Vilares Ferro and Bernard A Dion. Efficient incremental parsing for context-free languages. In *International Conference on Computer Languages*, pages 241–252, 1994.

[27] Bernd Fischer, Carsten Hammer, and Werner Struckmann. ALADIN: A scanner generator for incremental programming environments. *Software: Practice and Experience*, 22(11):1011–1025, 1992.

[28] C. N. Fischer, B. A. Dion, and J. Mauney. A Locally Least-Cost LR-Error Corrector. Technical Report 363, University of Wisconsin, August 1979.

[29] Charles N Fischer, Gregory F Johnson, Jon Mauney, Anil Pal, and Daniel L Stock. The Poe language-based editor project. In *ACM SIGSOFT Software Engineering Notes*, volume 9, pages 21–29. ACM, 1984.

[30] Karl Flinders. EDF blames system upgrade for customer complaint failures. `https://www.computerweekly.com/news/2240170222/EDF-blames-system-upgrade-for-customer-complaint-failures`, 2012. Accessed: 2018-08-01.

[31] Karl Flinders. Experian admits mobile and online issues after system upgrade. `https://www.computerweekly.com/news/450422339/Experian-admits-mobile-and-online-issues-after-system-upgrade`, 2017. Accessed: 2018-08-01.

[32] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL*, pages 111–122, Jan 2004.

[33] Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):58–70, 1979.

[34] Bryan Glick. The biggest surprise about TSB's IT disaster is that people are still surprised when banks' IT fails. `https://www.computerweekly.com/blog/Computer-Weekly-Editors-Blog/The-biggest-surprise-about-TSBs-IT-disaster-is-that-people-are-still-surprised-when-banks-IT-fails`, 2018. Accessed: 2018-08-01.

[35] Robert Grimm. Better extensibility through modular syntax. In *ACM SIGPLAN Notices*, volume 41, pages 38–51. ACM, 2006.

[36] Michael A Harrison and Vance Maverick. Presentation by tree transformation. In *Compcon*, pages 68–73, Sep 1997.

[37] Fahimeh Jalili and Jean H Gallier. Building friendly parsers. In *POPL*, pages 196–206, Jan 1982.

[38] S. C. Johnson. YACC: Yet Another Compiler-Compiler. Technical Report Comp. Sci. 32, Bell Labs, July 1975.

[39] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Generalised parsing: Some costs. In *International Conference on Compiler Construction*, pages 89–103. Springer, 2004.

[40] Lennart CL Kats, Karl T Kalleberg, and Eelco Visser. Domain-specific languages for composable editor plugins. *Electronic Notes in Theoretical Computer Science*, 253(7):149–163, 2010.

[41] Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, Oct 2010.

[42] Amir Ali Khwaja and Joseph E. Urban. Syntax-directed Editing Environments: Issues and Features. In *SAC*, pages 230–237, Feb 1993.

[43] Oleg Kiselyov. Differentiating parsers. `http://okmij.org/ftp/continuations/differentiating-parsers.html`, 2009. Accessed: 2018-01-04.

[44] Donald Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, Dec 1965.

[45] Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *SLE*, pages 311–331. Oct 2013.

[46] Thomas Kuhn and Olivier Thomann. Eclipse Corner: Abstract Syntax Tree. `https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`, 2006. Accessed: 2018-07-06.

[47] Ilya Lakhin. Papa Carlo. `http://lakhin.com/projects/papa-carlo/`, 2014. Accessed: 2018-01-04.

[48] Bernard Lang. On the usefulness of syntax directed editors. In *Advanced Programming Environments*, pages 47–51. Springer, 1986.

[49] J.-M. Larchevêque. Optimal incremental parsing. 17:1–15, 01 1995.

[50] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41, 1986.

[51] Clayton Lewis and Donald A Norman. Designing for error. In *Readings in Human–Computer Interaction*, pages 686–697. Elsevier, 1995.

[52] Warren X Li. A simple and efficient incremental LL (1) parsing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 399–404. Springer, 1995.

[53] Warren X Li. A new approach to incremental LR parsing. *J. Prog. Lang.*, 5(1):173–188, 1997.

[54] Scott McPeak and George C Necula. Elkhound: A fast, practical GLR parser generator. In *International Conference on Compiler Construction*, pages 73–88. Springer, 2004.

[55] Microsoft. The .NET Compiler Platform SDK. *https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/*, 2017. Accessed: 2019-03-21.

[56] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[57] Lisa Rubin Neal. *Cognition-sensitive design and user modeling for syntax-directed editors*, volume 18. ACM, 1987.

[58] Albert Ng, Michelle Annett, Paul Dietz, Anoop Gupta, and Walter F. Bischof. In the blink of an eye: investigating latency perception during stylus interaction. In *CHI*, 2014.

[59] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler Error Messages: What Can Help Novices? In *SIGCSE*, pages 168–172, March 2008.

[60] David Notkin. The GANDALF project. *Journal of Systems and Software*, 5(2):91–105, 1985.

[61] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In *ESOP*, pages 205–231. Springer, April 2015.

[62] David Pager. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7(3):249–268, 1977.

[63] Rohit J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, Oct 1966.

[64] Vaclav Pech, Alex Shatalin, and Markus Voelter. JetBrains MPS as a tool for extending Java. In *PPPJ*, pages 165–168, Sep 2013.

[65] Luigi Petrone. Reusing batch parsers as incremental parsers. pages 111–123, 1995.

[66] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. Language Boxes. In *SLE*, pages 274–293, Oct 2009.

[67] Pieter R Roelfsema, Damiaan Denys, and P Christiaan Klink. Mind Reading and Writing: The Future of Neurotechnology. *Trends in cognitive sciences*, 2018.

[68] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Not.*, 24(7):170–178, Jun 1989.

[69] Advait Sarkar. The impact of syntax colouring on program comprehension. In *Proceedings of the 26th annual conference of the psychology of programming interest group (ppig 2015)*, pages 49–58, 2015.

[70] August Schwerdfeger and Eric Van Wyk. Verifiable Composition of Deterministic Grammars. In *PLDI*, Jun 2009.

[71] Elizabeth Scott and Adrian Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177 – 189, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[72] John J. Shilling. Incremental LL (1) parsing in language-based editors. *IEEE transactions on software engineering*, 19(9):935–940, 1993.

[73] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on software engineering*, (5):595–609, 1984.

[74] Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, Sep 1981.

[75] Laurence Tratt and Lukas Diekmann. Grmtools. `https://github.com/softdevteam/grmtools`, 2019. Accessed: 2019-03-15.

[76] V Javier Traver. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, 2010, 2010.

[77] Michael L Van De Vanter. Practical language-based editing for software engineers. In *Workshop on Software Engineering and Human-Computer Interaction*, pages 251–267. Springer, 1994.

[78] Michael L Van De Vanter and Marat Boshernitsan. Displaying and editing source code in software engineering environments. In *Second International Symposium on Constructing Software Engineering Tools (CoSET'2000), 5 June 2000, Limerick Ireland*, 2000.

[79] Mark GJ Van den Brand, Jeroen Scheerder, Jurgen J Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *International Conference on Compiler Construction*, pages 143–158. Springer, 2002.

[80] Jan van Eijck. Let's accept rejects, but only after repairs. In *Liber Amicorum Paul Klint*, pages 117–128. November 2007.

[81] Guido Van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.

[82] Eric Van Wyk and August Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. In *Intl. Conf. on Generative Programming and Component Engineering, GPCE 2007*. ACM, October 2007.

[83] Naveneetha Vasudevan and Laurence Tratt. Detecting Ambiguity in Programming Language Grammars. In *SLE*, pages 157–176, Oct 2013.

[84] Jurgen Jordanus Vinju. *A Type-driven Approach to Concrete Meta Programming*. CWI. Software Engineering [SEN], 2005.

[85] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sep 1997.

[86] Eelco Visser et al. *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.

[87] Tim Wagner. Personal communication, February 2017.

[88] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. PhD thesis, University of California, Berkeley, Mar 1998.

[89] Tim A. Wagner and Susan L Graham. Incremental analysis of real programming languages. In *ACM SIGPLAN Notices*, volume 32, pages 31–43. ACM, 1997.

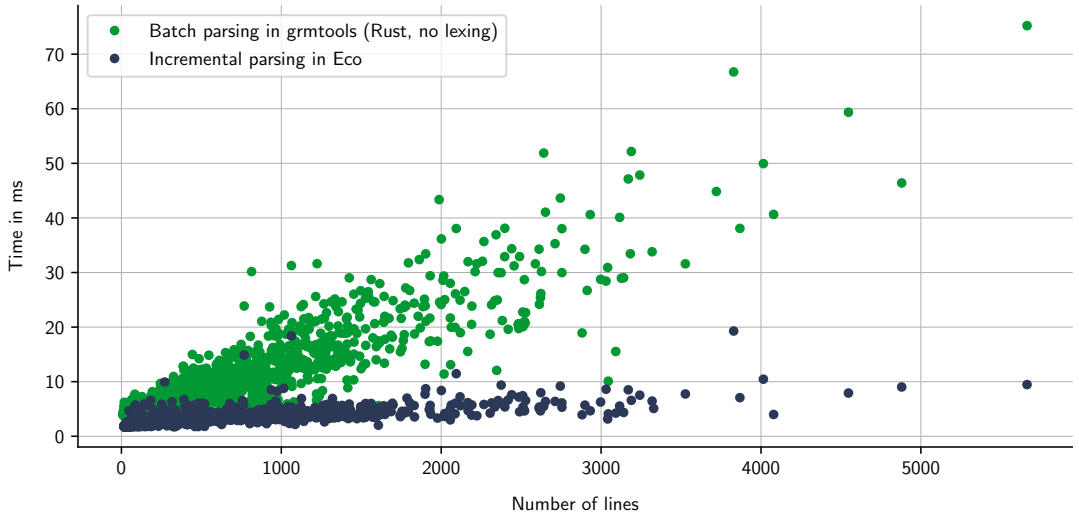
[90] Wuu Yang. An incremental LL (1) parsing algorithm. *Information Processing Letters*, 48(2):67–72, 1993.

[91] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ programs with meta-AspectJ. In *International Conference on Generative Programming and Component Engineering*, pages 1–18. Springer, 2004.

Appendix A

Incremental vs batch parsing performance without lexing

The following shows a comparison of incremental parsing in *Eco* with a batch parser generated using grmtools, which doesn't include lexing times. While the results of the batch parser are roughly two times faster than with lexing, incremental parsing in *Eco* is still about 5-7x faster.



Appendix B

Possible refinement problem

The following shows a scenario where Wagner's error recovery behaves unexpectedly and discusses a possible reason for it.

B.1 Current vs previous version in `pass2`

When implementing Wagner's refinement algorithm I discovered scenarios where a subtree would be retained when I was expecting the algorithm to run out-of-context analysis on it. Figure B.1 shows an example of such a scenario.

The figure shows the outcome that we would expect when isolating the subtree at node `S`, where `A` is discarded, `B` and `C` are retained, and `G` is processed via out-of-context analysis. However, even though `A` spans the error node in the previous version, it does not in the current version. Wagner's `pass2` method uses the current version of the tree to determine the offset and text-length of a node. In the current version `A` does not span the error location and thus is processed with `retain_or_discard` to process all retainable subtrees. `A` itself is not retainable, since its text-length is not the same as in the previous version, so its changes are discarded. Then `retain_or_discard` is recursively called on its children, of which `B` and `C` are retainable. The child `G` is not retainable, since it doesn't exist in the current version, nor are any of its children. In particular `G` is not processed with out-of-context analysis so any changes within it won't be parsed and integrated into the parse tree.

The reason for this is that Wagner uses the current version of a node's text-length within the `pass2` method. However, it doesn't make much sense to call `retain_or_discard` on `A` which due to its changed text-length can never be retained. This is true for any node that spans the error node in the previous version. In fact, the only subtrees we can retain

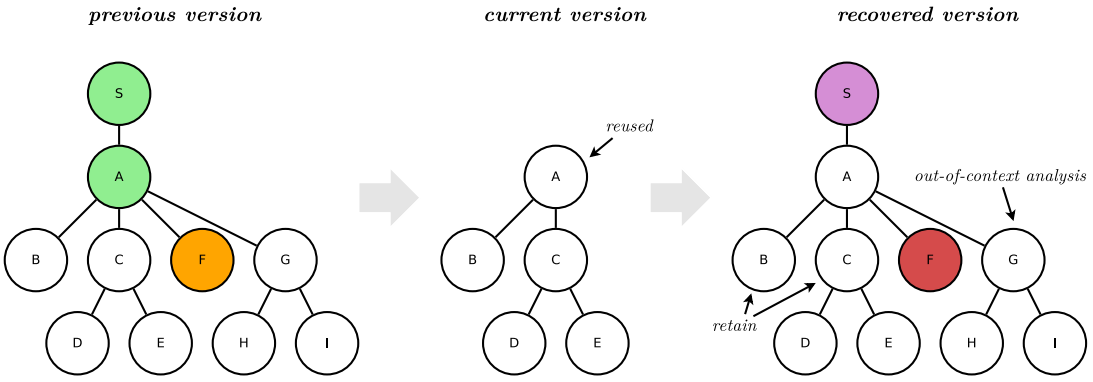


Figure B.1: An example where Wagner's refinement algorithm behaves unexpectedly. It shows error recovery on a parse tree as it is expected. The error occurs in node `F`, causing the isolation of node `S`. Node `B` and `C` can be retained and `G` is processed via out-of-context analysis. However, Wagner's refinement algorithm calls `retain_or_discard` on `A` which means `G` is not out-of-context analyses and thus any changes within it won't be integrated into the parse tree.

are the ones that have the same text-length in both versions and thus can never span the error node. If we instead recursively call `pass2` on `A` it would allow us to still retain `B` and `C`, but run out-of-context analysis on `G`. We can achieve this by simply using the previous instead of the current version in `pass2`.

In *Eco*'s implementation this change was briefly applied and an initial test suite run did not reveal any problems. However, it is difficult to say if this change affects error recovery in other unexpected ways. Since Wagner's version still works well in most cases, this change was thus reverted again, as it requires some more reasoning about its effects on the overall algorithm.

Appendix C

Incremental parsing algorithm

The following shows the full incremental parsing algorithm including all changes and improvements discussed in Chapter 3: setting the `exists`-flag when nodes get removed from or insert into the parse tree; using bottom-up node reuse upon reductions; skipping empty nonterminals in optimistic shifts; only revisiting isolated subtrees if they themselves or their surrounding context has changed.

```
state: int = 0
stack: List[Node] = []
reused: Set[Node] = set()

def incparse(bos: Node):
  verifying: bool = False
  la: Node = next_lookahead(bos)
  while True:
    if la is a terminal:
      action = parsetable.lookup(state, la.symbol)
      if action is Shift:
        verifying = False
        shift(la, action)
        la = next_lookahead(la)
      elif action is Reduce:
        reduce(action.production)
      elif action is Accept:
        return True
      elif action is Error:
        if verifying:
          right_breakdown()
          verifying = False
        else:
          la = recover()
    else: # la is a nonterminal
      if la.nested_changes or len(la.children) == 0 or la.has_errors() \
          or (la was isolated and surrounding_context(la).changed):
        la = left_breakdown(la)
      else:
        action = parsetable.lookup(state, la.symbol)
```

```
31          if action is Shift:
32            verifying = True
33            shift(la, action)
34            la = next_lookahead(la)
35          elif action is Reduce:
36            reduce(action.production)
37          elif action is Error:
38            la = left_breakdown(la)
39
40  def left_breakdown(la: Node) -> Node:
41    la.exists = False
42    if len(la.children) > 0:
43      return la.children[0]
44    else:
45      return next_lookahead(la)
46
47  def right_breakdown():
48    node = stack.pop() # remove optimistically shifted subtree
49    node.exists = False
50    while node is nonterminal:
51      reused.discard(node)
52      for c in node.children:
53        action = parsetable.lookup(stack[-1].state, c.symbol)
54        shift(c, action)
55      node = stack.pop()
56      node.exists = False
57    action = parsetable.lookup(state[-1].state, node.symbol)
58    shift(node, action) # leave final token on stack
59
60  def next_lookahead(la: Node) -> Node:
61    while la.right_sibling(prev) is None:
62      la = la.get_parent(prev)
63    return la.right_sibling(prev)
64
65  def shift(la: node, s: Shift):
66    stack.append(la)
67    la.exists = True
68    state = la.state = s.state
69
70  def reduce(p: Production):
71    children: List[Node] = []
72    for i in p.length():
73      children.append(stack.pop())
74    state = stack[-1].state
75    n = ambig_reuse_check(p, children)
76    goto = parsetable.lookup(state, n.symbol)
77    state = goto.state
78    stack.append(n)
79    n.exists = True
80    calc_text_length(n, current_version)
81    if p.has_rewriterule():
82      exec_rewriterule(n, p)
83
84  def surrounding_context(node: Node) -> Node:
85    la: Node = next_lookahead(node)
86    while la is a nonterminal:
```

```
87      if la.has_children():
88        la = la.children[0]
89      else:
90        la = next_lookahead(la)
91    return la
92
93  def calc_text_length(node: Node, version: int):
94    if node is terminal:
95      l = len(node.value(version))
96    else:
97      l = 0
98      for c in node.get_children(version):
99        l += c.text_length(version)
100   node.set_text_length(l, version)
101
102 def exec_rewriterule(node: Node, p: Production):
103   astnode: AstNode = p.rewriterule.execute(node)
104   if not is_reusable_astnode(node.ast, astnode):
105     node.ast = astnode
106
107 def is_reusable_astnode(old: AstNode, new: AstNode):
108   if old is None:
109     return False
110   if old.name != new.name:
111     return False
112   if children are not the same:
113     return False
114   return True
115
116 # Error recovery
117
118 def recover():
119     error_offset: int = stack_offset(stack[-1])
120     iso: Node, i: int = find_iso_tree()
121     stack = stack[:i+1]
122     refine(iso, error_offset)
123     stack.append(iso)
124     iso.exists = True
125     return next_lookahead(iso)
126
127 def find_iso_tree() -> (Node, int):
128   node = stack[-1]
129   while node is not root:
130     node = node.parent
131     offset: int = get_offset(node, curr)
132     sl = 0
133     for i in len(stack):
134       if sl == offset and can_shift(node, i):
135         return node, i
136       elif sl > offset:
137         break
138       sl += stack[i].text_length()
139   return root, 0
140
141 def can_shift(node: Node, i: int) -> bool:
142     s: int = stack[i].state
```

```
143     action = parsetable.lookup(s, node.symbol)
144     return action is Goto
145
146 def stack_offset() -> int:
147   l = 0
148   for n in stack:
149     l += n.text_length(curr)
150   return l
151
152 def get_offset(node: Node, version: int) -> int:
153   offset = 0
154   while node is not root:
155     left: Node = node.left\_sibling(version)
156     if left:
157       node = left
158       offset += node.text_length(version)
159     else:
160       node = node.get_parent(version)
161   return offset
162
163 def refine(isonode: Node, error_offset: int):
164     offset = stack_offset()
165     pass1(isonode, offset, offset, error_offset)
166     isonode.discard()
167     pass2(isonode, offset, error_offset)
168
169 def pass1(node: Node, offset: int, poffset: int, error_offset: int):
170   for child in node.get_children(prev):
171     if offset + child.text_length(curr) <= error_offset:
172       find_retainable(child, offset, poffset)
173     elif offset < error_offset:
174       pass1(child, offset, poffset)
175     else:
176       break
177     offset += child.text_length(curr)
178     poffset += child.text_length(prev)
179
180 def find_retainable(node: Node, offset: int, poffset: int):
181   if node.exists:
182     if not node.nested_changes() or same_text_pos(node, offset, poffset):
183       add node to retainable
184       return
185   for child in node.get_children(prev):
186     find_retainable_subtrees(node)
187     offset += child.text_length(curr)
188     poffset += child.text_length(prev)
189
190 def same_text_pos(node: Node, offset: int, poffset: int) -> bool:
191   if node.text_length(prev) == node.text_length(curr) and offset == poffset:
192     return True
193   return False
194
195 def pass2(node: Node, offset: int, error_offset: int):
196   for child in node.get_children(curr):
197     if offset > error_offset:
198       out_of_context_analysis(child)
```

```
199       elif offset + child.text_length(curr) <= error_offset:
200         retain_or_discard(child, node)
201       else:
202         child.discard()
203         pass2(child, offset)
204       offset += child.text_length(curr)
205
206   def retain_or_discard(node: Node, parent: Node):
207     if node in retainable:
208         node.set_parent(parent)
209         remove node from retainable
210       return
211     discard_and_mark(node)
212     for c in node.get_children(curr):
213       retain_or_discard(c, node)
214
215   def discard_and_mark(node: Node):
216     node.load(previous version)
217     if node.changed:
218       node.local_error = True
219     if node.nested_changes:
220       node.nested_errors = True
221     else:
222       node.nested_errors = False
223
224   def out-of-context-analysis(subtree: Node):
225     vbos: Node = preceding(subtree)
226     veos: Node = next_lookahead(subtree)
227     pstate: int = subtree.state
228     psymbol: Symbol = subtree.symbol
229     op = new incremental ooc parser
230     op.state = vbos.state
231     op.tree = Root(vbos, subtree, veos>)
232     if op.incparse(veos, pstate) and op.stack[0].symbol == psymbol:
233       integrate changes
234     else:
235       revert_changes(subtree)
236
237   def revert_changes(node: Node):
238     if node.has_changes():
239       node.load(self.prev_version)
240       if node.nested_changes:
241         node.nested_errors = True
242       if node.changed:
243         node.local_error = True
244       for c in node.children:
245         revert_changes(c)
```

Appendix D

Creating compositions in *Eco*

The following shows how two languages can be composed from start to finish using Eco. A composition generally consists of a base language and one or more embedded languages. In this example, we choose `ToyPHP` as our base language, whose grammar is shown in Figure D.2. As our embedded language we choose `ToyPython`, shown in Figure D.3. While it is possible to embed entire languages, in this case we only want to embed two subsets of `ToyPython`: functions and expressions. Figure D.1 shows an example of the resulting composition in action.

To create a composition, we use the same format (JSON) that is used to define languages in *Eco*. The composition of `ToyPHP` and `ToyPython` is shown below:

```
{
    "name": "ToyPHP + ToyPython",
    "file": "grammars/toyphp.eco",
    "base": "PHP",
    "compositions": [
        {
            "location": "add_expr",
            "name": "ToyPy expr",
            "file": "grammars/toypy.eco",
            "base": "Python",
            "subset": "mul_expr"
        },
        {
            "location": "function",
            "name": "ToyPy method",
            "file": "grammars/toypy.eco",
            "base": "Python",
            "subset": "method"
        }
    ]
}
```

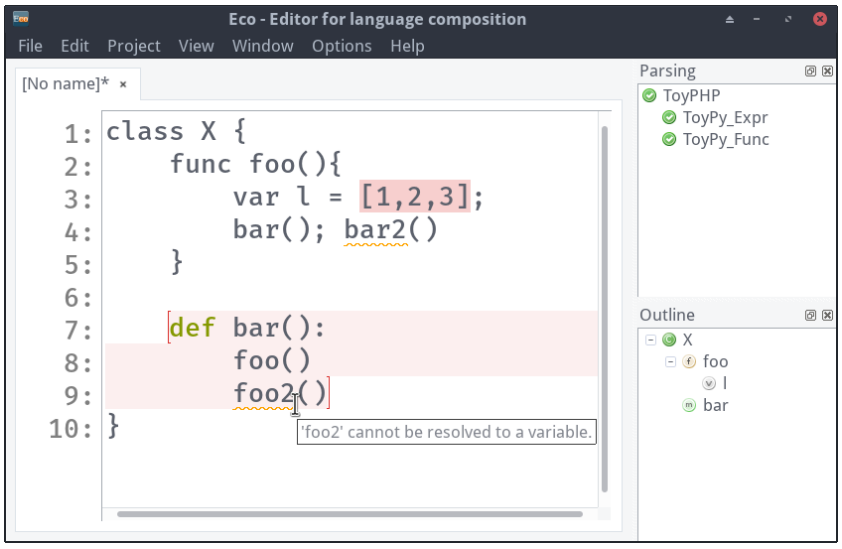


Figure D.1: Composition of two languages where a Python-like language `ToyPython` has been embedded into a PHP-like language `ToyPHP`. The user can write `ToyPython` code to embed either expressions or functions. Name resolution informs the user about undefined variables or functions both locally as well as across languages using orange, squiggly lines.

Each language definition has three required fields: the display name of the language (*name*), the location of its grammar (*file*), and which language the definition is based on (*base*), which *Eco* uses for syntax-highlighting and name binding. Optional fields are: *subset*, which is used to create subsets of languages; and *compositions*, which describes the languages we want to embed. The latter is a list of language definitions which require an additional field *location*, telling the outer language where we want the embedded language to be valid. For example, for the above composition of `ToyPHP` and `ToyPython`, we have embedded in the outer language the expressions subset of `ToyPython` at the location '`add_expr`', and the functions subset at the location '`function`'. Grammars are written within *Eco* itself, which it provides syntax highlighting and name binding to aid the user, and are stored as `.eco` files.

We can now load this composition into *Eco*, which makes it available as an option when creating or importing files. Automatic language box detection and insertion works out-of-the-box, though, while automatic detection is activated by default, automatic insertion needs to be activated via the editor's menu.

Adding name binding rules to our new composition, which allows the displaying of undeclared variables, requires some more work. Eco uses the NBL approach to define name binding rules using references to the AST nodes constructed by the rewrite rules during parsing. Eco automatically loads name binding rules of the same name as the grammar and the '`.nb`' file-extension. The name binding rules for `ToyPHP` and `ToyPython` can be found in Figure D.1.

```
class      ::= "class" "ID" "{" functions "}"
                                  {ClassDeclaration(name=#1, body=#3)};

functions  ::= functions function       {#0 + [#1]}
             | function                 {[#0]};

function   ::= "func" "ID" "(" ")" "{" statements "}"
                                  {FunctionDefinition(name=#1, body=#5)};

statements ::= statements statement     {#0 + [#1]}
             | statement                {[#0]};

statement  ::= assignment               {#0}
             | func_call                {#0};

func_call  ::= "ID" "(" ")"             {FunctionCall(name=#0)};

assignment ::= "var" "ID" "=" add_expr  {Assignment(name=#1, value=#3)};

add_expr   ::= scalar                   {#0}
             | add_expr "+" scalar      {Add(lhs=#0, rhs=#2)};

scalar     ::= "ID"                     {Var(name=#0)}
             | "INT"                    {Num(value=#0)};
%%
%implicit_ws=true
%%
INT:       "[0-9]+"
ID:        "[a-zA-Z_][a-zA-Z0-9]*"
<ws>:      "[ \\t]+"
<return>:  "[\\n\\r]"
```

Figure D.2: Grammar and lexing rules of `ToyPHP`, a small language syntactically inspired by PHP. The grammar has been extended with AST rewrite rules.

When a program with language boxes is parsed, each language box runs their own separate name binding analysis, confined to the boundaries of the box. The outer language then merges the results from the language boxes into its own results. However, `ToyPHP` and `ToyPython` use different names to define certain types, e.g. the former defines functions as `function`, while the latter uses `method`. For cross-language scoping to work as expected we need to adjust the scoping rules of each language to define which parts of another language it can 'see'. In this case, `ToyPHP`'s `ClassDeclaration` and `FunctionCall` rules were extended to also scope (or reference) `method`, while `ToyPython`'s `FunctionCall` now also references `function`. This allows both languages to see each other's function definitions as shown in Figure D.1.

```
class      ::= "class" "ID" ":" methods      {ClassDecl(name=#1, body=#3)};

methods    ::= methods method                {#0 + [#1]}
             | method                        {[#0]};

method     ::= "def" "ID" "(" ")" ":" stmts  {Method(name=#1, body=#5)};

stmts      ::= stmts statement               {#0 + [#1]}
             | statement                     {[#0]};

statement  ::= assignment                    {#0}
             | funccall                      {#0};

funccall   ::= "ID" "(" ")"                  {FuncCall(name=#0)};

assignment ::= "ID" "=" mul_expr             {Assign(var=#0, val=#2)};

mul_expr   ::= add_expr                      {#0}
             | mul_expr "*" add_expr         {Mul(l=#0, r=#2)};

add_expr   ::= atom                          {#0}
             | add_expr "+" atom             {Add(l=#0, r=#2)};

atom       ::= "ID"                          {Ident(name=#0)}
             | "NUM"                         {Int(val=#0)}
             | list                          {List{val=#0};
list       ::= "[" "]"                       {[]}
             | "[" list_items "]"            {#1};
list_items ::= atom                          {[#0]}
             | list_items "," atom           {#0 + [#2]};
%%
%implicit_ws=true
%%
NUM:       "[0-9]+"
ID:        "[a-zA-Z_][a-zA-Z0-9]*"
<ws>:      "[ \\t]+"
<return>: "[\\n\\r]"
```

Figure D.3: Grammar and lexing rules of `ToyPython`, a small language syntactically inspired by Python. The grammar has been extended with AST rewrite rules.

```
ClassDeclaration(name, body):
  defines class name
  scopes function, method
FunctionDefinition(name, body):
  defines function name
  scopes variable
Assignment(name, value):
  defines variable name
  references variable value
Identifier(name):
  references variable name
FunctionCall(name):
  references function, method name
```

(a) ToyPHP

```
ClassDecl(name, body):
  defines class name
  scopes method
Method(name, body):
  defines method name
  scopes variable
Assign(name, value):
  defines variable name
  references variable value
Ident(name):
  references variable name
FuncCall(name):
  references method, function name
```

(b) ToyPython

Listing D.1: Name binding rules for ToyPHP and ToyPython. Both rules are similar to most object-oriented languages, e.g. a class can see functions defined within it, while functions can see variables definitions within their body. To allow cross-language scoping, ToyPHP's classes and function calls are also allowed to see `method`, which is a type unique to ToyPython. Similarly, function calls in ToyPython are allowed to reference `function`. The additions are highlighted in green.

Appendix E

Additional examples of language boxes inside strings

The following contains some additional examples showing how the merging algorithm from Listing 6.3 handles different variations of the scenarios shown in Section 6.4.

E.1 Merging normal nodes into multinodes

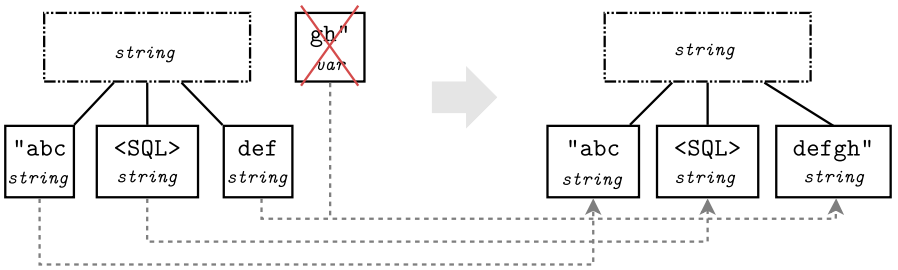
This example shows how a normal node can be merged into an existing multinode. Let's assume a scenario where a user moves the end quote of a string to include another token currently located next to the string:

```
"abc  <SQL>  def"gh   ⇒   "abc  <SQL>  defgh"
```

After re-lexing, the lexer returns the following generated tokens and processed nodes:

generated tokens	processed nodes
`(['"abc', lbox, 'defgh"'], string)`	`multinode` `gh"`

Merging of this token is very similar to what we have already seen in the examples in Section 6.4. First, the multinode is reused and its children are overwritten: the node `"abc` and the language box remain unchanged, while node `def` is updated to `defgh"`. Afterwards the excess node `gh"` is simply deleted. The following figure shows a summary of that process:



E.2 Creating a normal and multinode at the same time

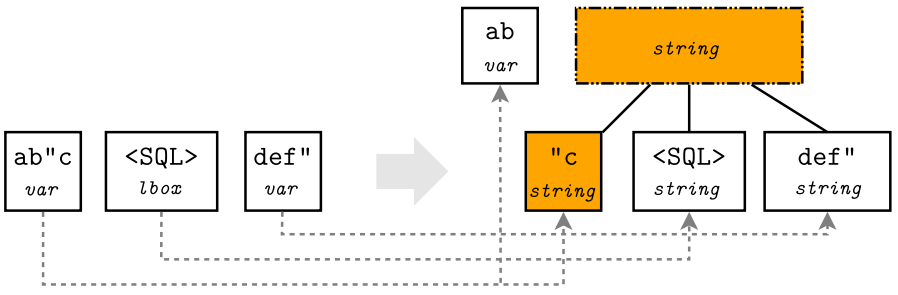
This example shows a scenario, where a node needs to be split up with one part remaining a normal node, while the other part needs to be moved inside of a multinode. Let's consider the following scenario, where the user inserted quotes into a program as shown here:

```
abc <SQL> def   ⇒   ab"c <SQL> def"
```

Re-lexing the changed nodes results into the following generated tokens and processed nodes:

generated tokens	processed nodes
(‘ab’, var) ([‘"c’, lbox, ‘def"’], string)	ab"c <SQL> def"

The merge algorithm starts with the generated token ‘ab’ and the processed node ab"c. Since the token is not part of a multitoken, no multinode needs to be created and we end up in the *Overwrite* branch of the algorithm, updating node ab"c's value to ‘ab’. The next generated token is ‘"c’, which is part of a multitoken. We thus first create a new multinode, and then use the *Insert* branch. Since the token is the first within the multitoken the function `insert_after` directly inserts it into the multinode (lines 39–40). The remaining tokens are merged using the *Overwrite* branch, updating existing nodes while also moving them into the multinode, as shown in the summary below:



E.3 Create a normal node from a multinode split

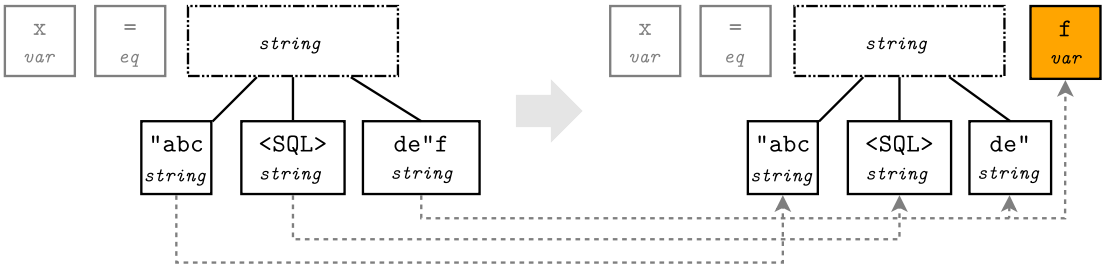
This example shows the reverse of the previous example. A string is manipulated in such a way, that a node that is part of a multinode needs to be split up, with one part remaining within the multinode while the other part is moved outside. Let's assume a user editing a program as follows:

`x = "abc <SQL> def"` ⇒ `x = "abc <SQL> de"f`

After re-lexing, this results in the following generated tokens and processed nodes:

generated tokens	processed nodes
`(['"abc', lbox, 'de"'], string)` `('f', var)`	`multinode`

The algorithm starts by overwriting the nodes `"abc`, `<SQL>`, and `def"f`, with the first two only updating their type and the last also changing its value to '`de"`'. After this, the next generated token (`new`) is '`f`', while `old` is set to `None`, and we end up in the *Insertion* branch of the algorithm. Since the previous token '`de"`' was the last child of the multitoken, `last` is set to the multinode and `current_mt` is set to `None` (lines 11–16). This results in token '`f`' simply being inserted after the multinode:



E.4 Applying changes within a multinode

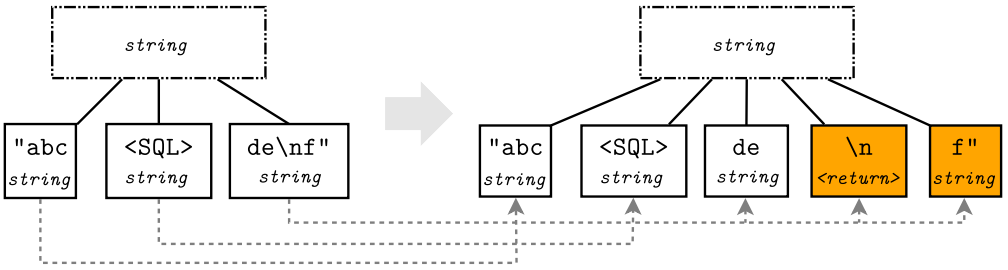
This example shows how changes are being applied within a multinode. This is a common scenario which happens when the user simply makes edits within a multinode. Most of the time this just results in extra tokens being inserted into the multinode. Though occasionally this can also lead to nodes being split up, as the following scenario shows. Here the user inserted a newline into a string. We also assume that as in *Eco*, newlines are treated specially, meaning that they always stand on their own and can't be combined with other text:

```
x = "abc  <SQL>  def"   ⇒   x = "abc  <SQL>  de\nf"
```

After re-lexing, this results in the following generated tokens and processed nodes:

generated tokens	**processed nodes**
`(['"abc', lbox, 'de', '\n', 'f"'], string)`	`multinode`

The algorithm begins by overwriting the nodes `"abc`, `<SQL` (even though they technically remain unchanged), and updating node `de\nf"` with '`de`'. With the next processed node `old` node being `None` and the next generated token `new` being '`\n`', the new token is simply inserted after `de`. Next up is another new token '`f"`', which is also simply inserted. Afterwards, both `new` and `old` are `None` and the algorithm terminates. The following figure shows a summary of those steps:



Appendix F

Automatic language boxes algorithm

The following summarises the algorithms shown in Chapter 7 and shows implementations for some of the less important helper functions, which are only explained in prose. Readers may find this summarised version helpful in understanding the algorithms that are otherwise split up over the chapter.

```
def post_incparse():
  for lang, start, end in candidates: # insert lbox
    replace_content_with_lbox(lang, start, end)

  for lbox in program: # remove lbox
    if can_remove_lbox(lbox):
        replace_lbox_with_content(lbox)

def find_candidates(error):
  # called during incremental parsing immediately after an error is found,
  # i.e. before recover()
  valid_boxes = []
  for lang in composition:
    cut = len(stack)
    while cut >= 0:
      if lbox of lang can be shifted at cut:
        valid_boxes.append((lang, cut))
      cut -= 1

  candidates = []
  for lang, cut in valid_boxes:
    results = consume_text(stack[cut], lang)
    for end in results:
      candidates.append((lang, cut, end))

  f = []
  for c in candidates:
    if confirm_candidate(c) and fixes_error(c, error):
        f.append(c)
  return f

```

```
32 def confirm_candidate(c):
33   parser = throwaway parser for outer language
34   lang, cut, end = c
35   parser.stack = stack[:cut]
36   parser.parse(newlbox(lang))
37   return can_parse_after(parser, end)
38
39 def fixes_error(c, error):
40   if error was parsed in is_valid:
41     return True
42   _, cut, end = c
43   start = stack[cut]
44   if start is error:
45     return True
46   while start is not end:
47     start = find_next_terminal(start)
48     if start is errornode:
49       return True
50   return False
51
52 def consume_text(node, lang):
53   lexer = lexer for lang starting at node
54   parser = parser for lang
55   results = []
56   token = lexer.next_token()
57   while token is not None:
58     if parser.parse(token):
59       if can_accept(parser):
60         results.append(lexer.last_node)
61     else: # parse error
62       break
63     token = lexer.next_token()
64   return results
65
66 def can_accept(parser):
67   tmp = parser.state[:]
68   accept = parser.parse(EOS)
69   parser.state = tmp
70   return accept
71
72 def can_remove_lbox(lbox):
73   parser = parser for outer language
74   preparse(get_bos(), lbox, parser)
75   results = consume_text(lbox.get_bos(), outer lang)
76   if results[-1] is last node in lbox:
77     if lbox contains parse errors:
78       return True
79     return can_parse_after(parser, lbox)
80
81 def can_parse_after(parser, node):
82   term = find_next_terminal(node)
83   while term is whitespace:
84     if not parser.parse(term):
85       return False
86     term = find_next_terminal(term)
87   return parser.parse(term)
```

Appendix G

Python grammar

A slightly modified Python grammar based on the grammar of CPython's reference implementation:

```
file_input :
      file_input "NEWLINE"
    | stmts
    | ;

stmts :
      stmts stmt
    | stmt ;

decorator :
      "@" dotted_name "(" arglist ")" "NEWLINE"
    | "@" dotted_name "(" ")" "NEWLINE"
    | "@" dotted_name "NEWLINE" ;

decorators :
      decorator
    | decorator decorators ;

decorated :
      decorators classdef
    | decorators funcdef ;

funcdef : "def" "NAME" parameters "COLON" suite ;

parameters :
      "(" varargslist ")"
    | "(" ")" ;

varargslist :
      fpdef_loop
    | fpdef_loop ","
    | fpdef_loop "," kwargs_opt
    |              kwargs_opt ;

fpdef_loop :
      fpdef_loop "," fpdef_opt
    | fpdef_opt ;
```

```
fpdef_opt :
      fpdef
    | fpdef "=" test ;

kwargs_opt :
      "*"  "NAME"
    | "**" "NAME"
    | "*"  "NAME" "**" "NAME" ;

fpdef ::=
      "NAME"
    | "(" fplist ")" ;

fplist :
      fpdef fplist_loop1 ","
    | fpdef fplist_loop1 ;

fplist_loop1 :
      fplist_loop1 "," fpdef
    | ;

stmt :
      simple_stmt
    | compound_stmt
    | slcomment "NEWLINE" ;

simple_stmt :
      small_stmt simple_stmt_loop1 ";" slcomment_opt "NEWLINE"
    | small_stmt simple_stmt_loop1     slcomment_opt "NEWLINE" ;

simple_stmt_loop1 :
      simple_stmt_loop1 ";" small_stmt
    | ;

small_stmt :
      expr_stmt
    | print_stmt
    | del_stmt
    | pass_stmt
    | flow_stmt
    | import_stmt
    | global_stmt
    | exec_stmt
    | assert_stmt ;

expr_stmt :
      testlist augassign yield_expr
    | testlist augassign testlist
    | expr_stmt_loop ;

expr_stmt_loop :
      expr_stmt_loop "=" testlist
    | expr_stmt_loop "=" yield_expr
    | testlist ;

augassign :
      "+=" | "-=" | "*=" | "/=" | "%=" | "&="
    | "|=" | "^=" | "<<=" | ">>=" | "**=" | "//=" ;

print_stmt :
      "print"
```

```
    | "print"      test print_stmt_loop1
    | "print"      test print_stmt_loop1 ","
    | "print" ">>" test
    | "print" ">>" test print_stmt_loop2
    | "print" ">>" test print_stmt_loop2 "," ;

print_stmt_loop1 :
      print_stmt_loop1 "," test
    | ;

print_stmt_loop2 :
      "," test
    | print_stmt_loop2 "," test ;

del_stmt : "del" exprlist ;

pass_stmt : "pass" ;

flow_stmt :
      break_stmt
    | continue_stmt
    | return_stmt
    | raise_stmt
    | yield_stmt ;

break_stmt : "break" ;

continue_stmt : "continue" ;

return_stmt :
      "return"
    | "return" testlist ;

yield_stmt : yield_expr ;

raise_stmt :
      "raise"
    | "raise" test
    | "raise" test "," test
    | "raise" test "," test "," test ;

import_stmt :
      import_name
    | import_from ;

import_name : "import" dotted_as_names ;

import_from : "from" import_option1 "import" import_option2 ;

import_option1 :
      dotted_name
    | dot_loop dotted_name
    | dot_loop ;

dot_loop :
      dot_loop "DOT"
    |          "DOT" ;

import_option2 :
      "*"
    | "(" import_as_names ")"
```

```
    | import_as_names ;

import_as_name :
      "NAME"
    | "NAME" "as" "NAME";

dotted_as_name :
      dotted_name
    | dotted_name "as" "NAME" ;

import_as_names :
      import_as_name import_as_names_loop1
    | import_as_name import_as_names_loop1 "," ;

import_as_names_loop1 :
      import_as_names_loop1 "," import_as_name
    | ;

dotted_as_names :
      dotted_as_name
    | dotted_as_names "," dotted_as_name ;

dotted_name ::= "NAME" dotted_name_loop1 ;

dotted_name_loop1 :
      dotted_name_loop1 "DOT" "NAME"
    | ;

global_stmt : "global" "NAME" global_stmt_loop1 ;

global_stmt_loop1 :
      global_stmt_loop1 "," "NAME"
    | ;

exec_stmt :
      "exec" expr
    | "exec" expr "in" test
    | "exec" expr "in" test "," test ;

assert_stmt :
      "assert" test
    | "assert" test "," test ;

compound_stmt :
      if_stmt | while_stmt | for_stmt | try_stmt
    | with_stmt | funcdef | classdef | decorated ;

if_stmt :
      "if" test "COLON" suite if_stmt_loop1
    | "if" test "COLON" suite if_stmt_loop1 "else" "COLON" suite ;

if_stmt_loop1 :
      if_stmt_loop1 "elif" test "COLON" suite
    | ;

while_stmt :
      "while" test "COLON" suite
    | "while" test "COLON" suite "else" "COLON" suite ;

for_stmt :
      "for" for_exprlist "in" testlist "COLON" suite
```

```
    | "for" for_exprlist "in" testlist "COLON" suite "else" "COLON" suite ;

try_stmt :
      "try" "COLON" suite "finally" "COLON" suite
    | "try" "COLON" suite try_stmt_loop1
    | "try" "COLON" suite try_stmt_loop1 "else" "COLON" suite
    | "try" "COLON" suite try_stmt_loop1 "finally" "COLON" suite
    | "try" "COLON" suite try_stmt_loop1
      "else" "COLON" suite "finally" "COLON" suite ;

try_stmt_loop1 :
      try_stmt_loop1 except_clause "COLON" suite
    |                except_clause "COLON" suite ;

with_stmt : "with" with_item with_stmt_loop1  "COLON" suite ;

with_stmt_loop1 :
      with_stmt_loop1 "," with_item
    | ;

with_item :
      test
    | test "as" expr ;

except_clause :
      "except"
    | "except" test
    | "except" test "as" test
    | "except" test "," test ;

suite :
      simple_stmt
    | slcomment_opt "NEWLINE" "INDENT" suite_loop "DEDENT" ;

suite_loop :
      suite_loop stmt
    | stmt;

testlist_safe :
      old_test
    | old_test testlist_safe_loop1
    | old_test testlist_safe_loop1 "," ;

testlist_safe_loop1 :
      testlist_safe_loop1 "," old_test
    |                     "," old_test ;

old_test :
      or_test
    | old_lambdef ;

old_lambdef :
      "lambda"             "COLON" old_test
    | "lambda" varargslist "COLON" old_test ;

test :
      or_test
    | or_test "if" or_test "else" test
    | lambdef;

or_test :
```

```
      and_test
    | or_test "or" and_test ;

and_test :
      not_test
    | and_test "and" not_test ;

not_test :
      "not" not_test
    | comparison ;

comparison : expr
           | comparison comp_op expr ;

comp_op :
      "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
    | "in" | "not" "in" | "is" | "is" "not" ;

expr :
      xor_expr
    | expr "|" xor_expr ;

xor_expr :
      and_expr
    | xor_expr "^" and_expr  ;

and_expr :
      shift_expr
    | and_expr "&" shift_expr ;

shift_expr :
      arith_expr
    | shift_expr "<<" arith_expr
    | shift_expr ">>" arith_expr ;

arith_expr :
      term
    | arith_expr "+" term
    | arith_expr "-" term ;

term :
      factor
    | term "*"  factor
    | term "/"  factor
    | term "%"  factor
    | term "//" factor ;

factor :
      "+" factor
    | "-" factor
    | "~" factor
    | power ;

power :
      atom
    | atom            "**" factor
    | atom power_loop
    | atom power_loop "**" factor ;

power_loop :
      power_loop trailer
```

```
    | trailer ;

atom :
      "("                 ")"
    | "(" yield_expr      ")"
    | "(" testlist_comp   ")"
    | "["                 "]"
    | "[" listmaker       "]"
    | "{"                 "}"
    | "{" dictorsetmaker "}"
    | "`" testlist1 "`"
    | "NAME"
    | number
    | atom_loop
    | multiline_string ;

number : "NUMBER" | "HEX" | "OCTAL" | "BINARY" ;

atom_loop :
      atom_loop single_string
    |           single_string ;

multiline_string : "MLS";
single_string : "dstring" | "sstring";

listmaker :
      test list_for
    | test listmaker_loop
    | test listmaker_loop "," ;

listmaker_loop :
      listmaker_loop "," test
    | ;

testlist_comp :
      test comp_for
    | test testlist_comp_loop
    | test testlist_comp_loop "," ;

testlist_comp_loop :
      testlist_comp_loop "," test
    | ;

lambdef :
      "lambda"             "COLON" test
    | "lambda" varargslist "COLON" test ;

trailer :
      "("         ")"
    | "(" arglist ")"
    | "[" subscriptlist "]"
    | "DOT" "NAME" ;

subscriptlist :
      subscript subscriptlist_loop
    | subscript subscriptlist_loop "," ;

subscriptlist_loop :
      subscriptlist_loop "," subscript
    | ;
```

```
subscript :
      "DOT" "DOT" "DOT"
    | test
    |      "COLON"
    |      "COLON" test
    |      "COLON"      sliceop
    |      "COLON" test sliceop
    | test "COLON"
    | test "COLON" test
    | test "COLON"      sliceop
    | test "COLON" test sliceop ;

sliceop :
      "COLON"
    | "COLON" test ;

exprlist :
      exprlist_loop
    | exprlist_loop "," ;

exprlist_loop :
      expr
    |  exprlist_loop "," expr ;

testlist :
      testlist_loop
    | testlist_loop "," ;

testlist_loop :
      testlist_loop "," test
    | test;

dictorsetmaker :
      test "COLON" test comp_for
    | test "COLON" test          dictorsetmaker_loop1
    | test "COLON" test          dictorsetmaker_loop1 ","
    | test comp_for
    | test          dictorsetmaker_loop2
    | test          dictorsetmaker_loop2 "," ;

dictorsetmaker_loop1 :
      dictorsetmaker_loop1 "," test "COLON" test
    | ;

dictorsetmaker_loop2 :
      dictorsetmaker_loop2 "," test
    | ;

classdef :
      "class" "NAME"                  "COLON" suite
    | "class" "NAME" "("          ")" "COLON" suite
    | "class" "NAME" "(" testlist ")" "COLON" suite ;

arglist :
      arglist_loop1 argument
    | arglist_loop1 argument ","
    | arglist_loop1 "*" test arglist_loop2
    | arglist_loop1 "*" test arglist_loop2 "," "**" test
    | arglist_loop1 "**" test ;

arglist_loop1 :
```

```
      arglist_loop1 argument ","
    | ;

arglist_loop2 :
      arglist_loop2 "," argument
    | ;

argument :
      test
    | test comp_for
    | test "=" test ;

list_iter :
      list_for
    | list_if ;

list_for :
      "for" for_exprlist "in" testlist_safe
    | "for" for_exprlist "in" testlist_safe list_iter ;

for_exprlist : exprlist ;

list_if :
      "if" old_test
    | "if" old_test list_iter ;

comp_iter :
      comp_for
    | comp_if ;

comp_for :
      "for" exprlist "in" or_test
    | "for" exprlist "in" or_test comp_iter ;

comp_if :
      "if" old_test
    | "if" old_test comp_iter ;

testlist1 : test testlist1_loop ;

testlist1_loop :
      testlist1_loop "," test
    | ;

yield_expr :
      "yield"
    | "yield" testlist ;

slcomment_opt :
      slcomment
    | ;

slcomment : "slcomment" ;
```

Appendix H

Glossary

Abstract Syntax Tree (AST) Simplified version of the parse tree that removes irrelevant information such as whitespace, brackets. Used to perform further analysis of the program, e.g. name binding, code generation.

Context-free grammar (CFG) A formal grammar consisting of a set of rules describing a formal language.

Domain specific language (DSL) A language designed for a specific problem (e.g. SQL for databases), unlike general purpose languages.

Error recovery A technique that allows a parser to continue parsing after a parsing error occured. There are various forms of error recovery, typically falling into two subsets: repairing and non-repairing.

Foreign function interface (FFI) Mechanism that allows a program to call functions written in another language.

Generalised parsing An extension of an LR parser that allows the parsing of ambiguous and nondeterministic grammars.

Graph An automaton generated by a parser generator from a grammar. Used as an intermediate representation to construct a parse table from.

Integrated Development Environment (IDE) A software development environment which typically includes build-automation tools, a debugger, and quality-of-life features, such as name binding, type analysis, code formatting, etc.

Lexing The process of taking and input string and splitting it up into several tokens according to a set of lexing rules.

Isolation Wagner's error recovery solution. Isolating a subtree allows an incremental parser to continue parsing by simply ignoring the changes within the isolated subtree.

Language box A special token that allows the embedding of one language into another.

Lookahead The amount of characters exceeding a token that the lexer needed to scan to generate the token. Also used to describe the next node that is being processed by the parser.

Lookback The amount of preceeding tokens whose type is dependent on another token. Can be calculated from lookahead values.

LR A popular type of bottom-up parsing which accepts deterministic CFGs, typically constructed using a parser generator.

Jetbrains MPS A modern syntax-directed editor used to design domain-specific languages.

Multinode A special parsetree nodes which looks like a token but can contain multiple tokens as children. Used to represent tokens which also contain language boxes, such as comments or strings.

Node An element in a parse tree representing nonterminals and tokens.

Nonterminal A node representing a grammar rule, generated by the parser during a reduction.

Offset The textual offset of a node within a parse tree.

Parsing The process of reading some input to determine if it is valid according to a grammar, while execution parsing actions (defined in the grammar) or constructing a parse tree.

Parse tree The tree representation of some input generated by the parser. Consists of nonterminals and tokens.

Parsing Expression Grammar (PEG) A grammar that is closed under composition and inherently umambiguous (ambiguities are solved using a choice operator which always select the first match).

Production Part of a grammar rule, also called alternative.

Recogniser A parser whose only goal is to determine if some input is valid, but does not generate a parse tree or run any other executions during parsing.

Rule Part of a grammar, which describes a feature of the language.

Scannerless parsing A form of generalised parsing where the tokenisation and parsing are done in a single step.

Syntax-directed editing (SDE) A form of editing programs which, unlike traditional text-editors, works on an AST and constructs programs from templates which can be filled in by the user, avoiding the need to parse user input and thus avoiding syntax errors.

Subtree A partial parse tree describing the tree under a specific node that is not the root node.

Symbol The type of a node. The most common type is nonterminal and terminal, although there are special types, like the beginning/end-of-stream as well as the end-of-file symbol.

Parse table Simplified, lightweight representation of a graph which is used by the parser to decide if some input is valid.

Terminal symbol Used by the parser to shift into different states. This is typically the type of a token, however there are also special terminal symbols, like the end-of-file symbol.

Text-length The textual length of a node in the parse tree. A token's text-length is the character length of its value, while the text-length of a nonterminal is the combined text-length of all tokens contained in its subtree.

Token Returned by the lexer when splitting up the user input according to some lexing rules. Each token represents a substring of the input and has a value, e.g. 1, and a type, e.g. INT.

Whitespace Characters such as spaces, tabs, and newlines.

Yacc Popular parser generator for LR grammars.